Lista 3

tags: `SO`

:::warning **xterm font resize:** `xterm -fa 'Monospace' -fs 14` :::

Zadanie 1

Zadanie 1. Zaprezentuj sytuację, w której proces zostanie **osierocony**. Uruchom po emulatora terminala przy pomocy polecenia «`xterm -e 'bash -i'`». W nowej pov 1000» jako **zadanie drugoplanowe** i sprawdź, kto jest jego rodzicem. Polecenier «SIGKILL» do uruchomionej wcześniej powłoki i sprawdź, kto stał się nowym rodz Zauważ, że powłoka jest **liderem sesji**. Co się dzieje z **sesją**, która utraci **terminal s** eksperyment wysyłając «SIGKILL» do emulatora terminala zamiast do powłoki.

Wskazówka: Obserwuj interakcję systemu z powłoką przy pomocy polecenia «`strace -e trace=s`

::: info

- **Proces osierocony** – proces, który utracił swojego rodzica. Zostaje on podpięty pod inny proces, najczęściej `init`.
- **Zadanie drugoplanowe** – zadanie, które jest wykonywane w tle, umożliwiając dalszą komunikację z powłoką przez terminal.
- **Sesja** – zbiór grup procesów podpiętych do jednego terminala sterującego `tty`.
- **Lider sesji** – proces rozpoczynający sesję, zazwyczaj powłoka.
- **Terminal sterujący** – terminal, z którego sesja pobiera i do którego wysyła informacje. Terminal może być terminalem sterującym tylko dla jednej sesji jednocześnie. Sesja może zmienić terminal sterujący, jednak w praktyce dzieje się to tylko podczas logowania użytkownika do systemu. :::

Prezentacja powstawania sierot

> Otwarcie nowego okna terminala z bashem `xterm -e 'bash -i'`
>
> W nowym terminalu: `sleep 1000 &` `ps -q [pid] -o pid,ppid` `kill -SIGKILL [ppid]`
>
> W pierwotnym terminalu: `ps -eo pid,ppid,command | grep sleep` `ps -eo pid,ppid,command | grep [ppid] | less`

`sleep` został podczepiony pod `init`

Co się dzieje z sesją, która traci terminal sterujący

> Nowe okno `xterm -e 'bash -i' &`
>
> W starym oknie `ps -e | grep -e 'bash\|xterm'` `strace -e trace=signal -p [bash pid]`
>
> W nowym oknie `kill -SIGKILL [xterm pid]`

Zadanie 2

Zadanie 2. Zapoznaj się z paragrafami „*Canonical Mode Input Processing*”, „*Specia Data and Output Processing*” podręcznika `termios(4)`. Jak zachowuje się sterownik w **trybie kanonicznym**? Posługując się rysunkiem [1, 62-1] wytłumacz w jaki spos (w tym kody sterujące) wchodzące do **kolejki wejściowej** i **kolejki wyjściowej**. Jak powinien zmienić program na czas wpisywania hasła przez użytkownika? Czemu e konfigurują sterownik terminala do pracy w trybie niekanonicznym?

Canonical Mode Input Processing

```
 In canonical mode input processing, terminal input is processed in units
of lines.  A line is delimited by a newline \n character, an end-of-
file (EOF) character, or an end-of-line (EOL) character.  See the Special
Characters section for more information on EOF and EOL.  This means that
a read request will not return until an entire line has been typed, or a
signal has been received.  Also, no matter how many bytes are requested
in the read call, at most one line is returned.  It is not, however, nec-
essary to read a whole line at once; any number of bytes, even one, may
be requested in a read without losing information.


{MAX_CANON} is a limit on the number of bytes in a line.  The behavior of
the system when this limit is exceeded is the same as when the input
queue limit {MAX_INPUT}, is exceeded.

Erase and kill processing occur when either of two special characters,
the ERASE and KILL characters (see the Special Characters section), is
received.  This processing affects data in the input queue that has not
yet been delimited by a newline NL, EOF, or EOL character.  This un-
delimited data makes up the current line.  The ERASE character deletes
the last character in the current line, if there is any.  The KILL char-
acter deletes all data in the current line, if there is any.  The ERASE
and KILL characters have no effect if there is no data in the current
line.  The ERASE and KILL characters themselves are not placed in the
input queue.
```

Special Characters

```
 Certain characters have special functions on input or output or both.
These functions are summarized as follows:

INTR    Special character on input and is recognized if the ISIG flag
        (see the Local Modes section) is enabled.  Generates a SIGINT
        signal which is sent to all processes in the foreground process
```

group for which the terminal is the controlling terminal. If ISIG is set, the INTR character is discarded when processed.

QUIT Special character on input and is recognized if the ISIG flag is enabled. Generates a SIGQUIT signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the QUIT character is discarded when processed.

ERASE Special character on input and is recognized if the ICANON flag is set. Erases the last character in the current line; see Canonical Mode Input Processing. It does not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the ERASE character is discarded when processed.

KILL Special character on input and is recognized if the ICANON flag is set. Deletes the entire line, as delimited by a NL, EOF, or EOL character. If ICANON is set, the KILL character is discarded when processed.

EOF Special character on input and is recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process, without waiting for a newline, and the EOF is discarded. Thus, if there are no bytes waiting (that is, the EOF occurred at the beginning of a line), a byte count of zero is returned from the read(2), representing an end-of-file indication. If ICANON is set, the EOF character is discarded when processed.

NL Special character on input and is recognized if the ICANON flag is set. It is the line delimiter `\n'.

EOL Special character on input and is recognized if the ICANON flag is set. Is an additional line delimiter, like NL.

SUSP If the ISIG flag is enabled, receipt of the SUSP character causes a SIGTSTP signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character is discarded when processed.

STOP Special character on both input and output and is recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to temporarily suspend output. It is useful with fast terminals to prevent output from disappearing before it can be read. If IXON is set, the STOP character is discarded when processed.

START Special character on both input and output and is recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to resume output that has been suspended by a STOP character. If IXON is set, the START character is discarded when processed.

CR Special character on input and is recognized if the ICANON flag is set; it is the `\r', as denoted in the C Standard {2}. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into a NL, and has the same effect as a NL character.

The following special characters are extensions defined by this system and are not a part of IEEE Std 1003.1 (``POSIX.1'') termios.

EOL2 Secondary EOL character. Same function as EOL.

WERASE Special character on input and is recognized if the ICANON flag is set. Erases the last word in the current line according to one of two algorithms. If the ALTWERASE flag is not set, first any preceding whitespace is erased, and then the maximal sequence of non-whitespace characters. If ALTWERASE is set, first any preceding whitespace is erased, and then the maximal sequence of alphabetic/underscores or non alphabetic/underscores. As a special case in this second algorithm, the first previous non-whitespace character is skipped in determining whether the preceding word is a sequence of alphabetic/underscores. This sounds confusing but turns out to be quite practical.

REPRINT
Special character on input and is recognized if the ICANON flag is set. Causes the current input edit line to be retyped.

DSUSP Has similar actions to the SUSP character, except that the SIGTSTP signal is delivered when one of the processes in the foreground process group issues a read(2) to the controlling terminal.

LNEXT Special character on input and is recognized if the IEXTEN flag is set. Receipt of this character causes the next character to be taken literally.

DISCARD
Special character on input and is recognized if the IEXTEN flag is set. Receipt of this character toggles the flushing of terminal output.

STATUS Special character on input and is recognized if the ICANON flag is set. Receipt of this character causes a SIGINFO signal to be sent to the foreground process group of the terminal. Also, if the NOKERNINFO flag is not set, it causes the kernel to write a status message to the terminal that displays the current load average, the name of the command in the foreground, its process ID, the symbolic wait channel, the number of user and system seconds used, the percentage of CPU the process is getting, and the resident set size of the process.

The NL and CR characters cannot be changed. The values for all the

```
The NL and CR characters cannot be changed.  The values for all the
remaining characters can be set and are described later in the document
under Special Control Characters.

Special character functions associated with changeable special control
characters can be disabled individually by setting their value to
{_POSIX_VDISABLE}; see Special Control Characters.

If two or more special characters have the same value, the function per-
formed when that character is received is undefined.
```

Writing Data and Output Processing

```
When a process writes one or more bytes to a terminal device file, they
are processed according to the c_oflag field (see the Output Modes sec-
tion). The implementation may provide a buffering mechanism; as such,
when a call to write(2) completes, all of the bytes written have been
scheduled for transmission to the device, but the transmission will not
necessarily have been completed.
```

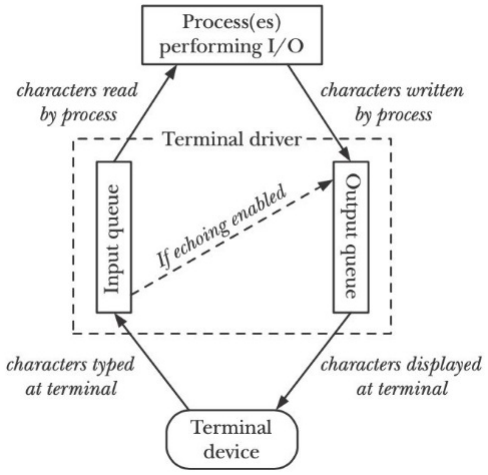
::: info **kolejka wejściowa** – znaki czytane z wejścia oczekujące na przejścia do procesu lub do kolejki wyjściowej

kolejka wyjściowa – znaki czekające na wydrukowanie w oknie terminala :::

Tryb kanoniczny

Wejście terminala jest przetwarzane liniami, kończonymi znakiem nowej lini (\n, EOL) lub końca pliku (Ctrl+D, EOF).

- Jednostką na wejściu jest zakończona znakiem EOL, NL, albo EOF linia
- Operacja czytania nie zakończy się dopóki nie zostanie wprowadzony znak kończący linię, lub nie zostanie odebrany sygnał.
- Można wczytać mniej bajtów niż jest w lini, pozostała część lini nie zostanie utracona (można ją wczytać kolejnym readem).
- W jednej lini nie może znaleźć się więcej bajtów niż zdefinowane jest w MAX_CANNON
- Sterownik terminala po napotkaniu specjalnych znaków może przeprowadzać operacje na danych już znajdujących się w kolejce wejściowej, ale jeszcze nie zakończonych znakiem końca lini. Są to np. znaki kill i erase, o których więcej powiedziane będzie w następnym zadaniu. Znaki te nie są umieszczane w kolejce.



Terminal przekazuje otrzymane znaki do kolejki wejściowej Znaki po wejściu do kolejki mogą przejść dwie drogi

- Natychchmiast po wpisaniu znaku do kolejki wejściowej trafia on do kolejki wyjściowej (włączone echo)
- Znaki są odczytywane pr rzez proces a następnie proces zwraca sekwencję znaków, które chce nam zwrócić na wyjście. Przykładowo przy wpisywaniu hasła nie chcemy wypisywać hasła na ekran, a po wpisaniu hasła przy danej operacji chcemy tylko by sam proces nam coś wypisał na ekran (echo wyłączone)
- Z kolejki wyjściowej znaki są już kolejno wyświetlane w terminalu.

Kolejki

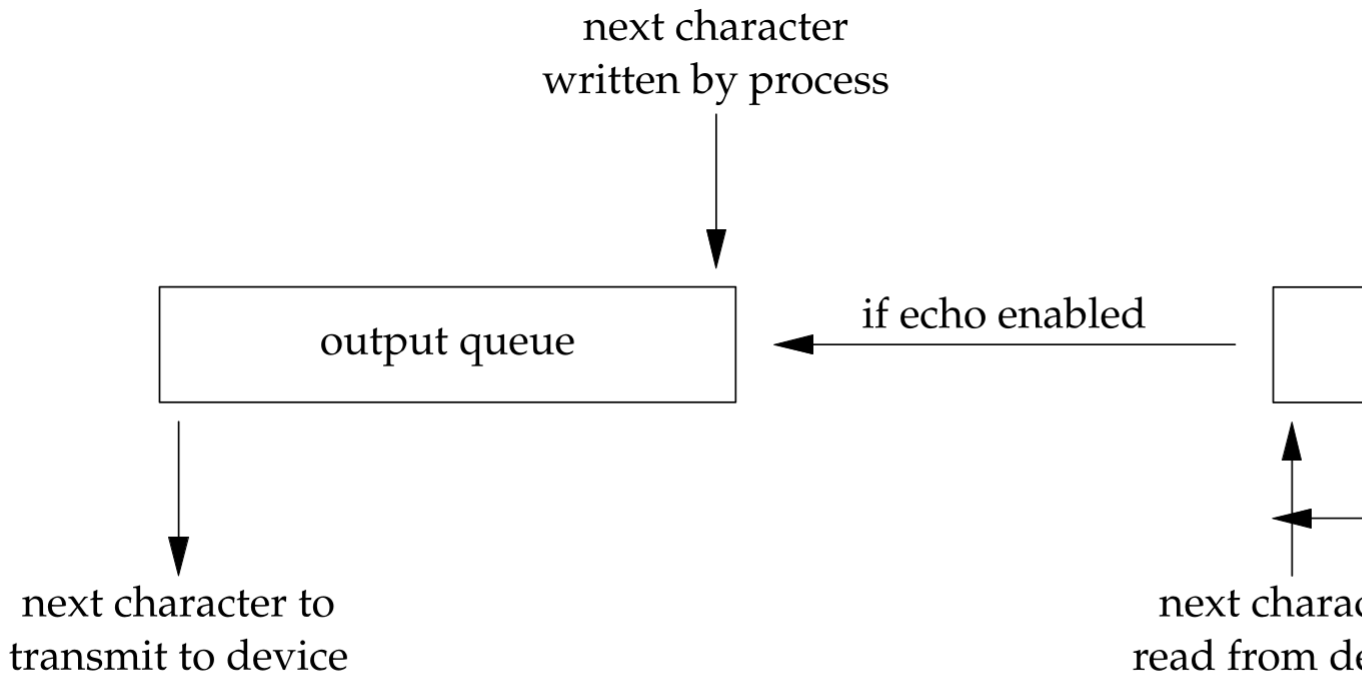


Figure 18.1 Logical picture of input and output queues fo

Jak zmienić konfigurację przy wpisywaniu hasła?

```
bash stty -echo
```

Czemu edytory takie jak vi używają trybu niekanonicznego?

Noncanonical Mode Input Processing

```
 In noncanonical mode input processing, input bytes are not assembled into
lines, and erase and kill processing does not occur.  The values of the
VMIN and VTIME members of the c_cc array are used to determine how to
process the bytes received.

VMIN represents the minimum number of bytes that should be received when
the read(2) system call successfully returns.  VTIME is a timer of 0.1
second granularity that is used to time out bursty and short term data
transmissions.  If VMIN is greater than {MAX_INPUT}, the response to the
request is undefined.  The four possible values for VMIN and VTIME and
their interactions are described below.
```

Zadanie 3

Zadanie 3. Wyświetl konfigurację terminala przy pomocy polecenia «`stty -a`». Wska terminala: zamienia na sygnały związane z **zarządzaniem zadaniami**, służą do **ed** może zostać poinformowany o zmianie **rozmiaru okna** terminala. W tym celu musi obsługi sygnału – którego? Jaką procedurą można wczytać nowy rozmiar okna?

::: info **Edycja wiersza** - zmiana na poziomie wejścia w terminalu (wyczyszczenie wiersza, skasowanie znaku itd.)

Zarządzanie zadaniami - kontrola sesji

Rozmiar okna - rozmiar terminalu liczony w kolumnach/wierszach. :::

```
stty -a
```

::: warning **Zarządzanie zadaniami:**

- intr(^C) - wysyła sygnał przerwania (`SIGINT`)
- quit(^\) - wysyła sygnał zamknięcia (`SIGQUIT`)
- swtch - zmienia na inną warstwę powłoki
- start(^Q) - resetuje wyjście po zatrzymaniu
- stop(^S) - zatrzymuje wyjście
- susp(^Z) - wysyła sygnał stopu (`SIGSTOP`)

Edycja wiersza:

- erase(^?) - usuwa ostatni wprowadzony znak
- kill(^U) - usunie obecny wiersz
- eof(^D) - znak końca pliku
- eol/eol2 - znak końca wiersza
- rprnt(^R) - powtórzy aktualny wiersz
- werase(^W) - usunie ostatnio wprowadzone słowo
- lnext(^V) - wprowadzi następny znak w cudzysłowie
- discard(^O) - włączenie discardowania wyjścia :::

Jak zrobić procedurę łapiącą sygnał zmiany okna?

```c
#include <sys/ioctl.h>
#include "include/csapp.h"


/*
    SIGWINCH - sygnał zmiany rozmiaru okna
    TIOCGWINSZ - typ requesta, w tym przypadku prosimy o rozmiar okna
*/


void sigwinch_handler(int sig)
{
    /* pobieramy wymiary okna structem winsize

            struct winsize {
                unsigned short ws_row;
                unsigned short ws_col;
                unsigned short ws_xpixel;
                unsigned short ws_ypixel;
            }
    */

    struct winsize window;
    ioctl(STDIN_FILENO, TIOCGWINSZ, &window);
    printf("row: %d col: %d\n", window.ws_row, window.ws_col);
}

int main()
{
    signal(SIGWINCH, sigwinch_handler);
    while (1) pause();
}
```

Zadanie 4

Zadanie 4. Urządzenie terminala zajmuje się interpretacją znaków i sekwencji steru
od sterownika terminala, oraz przetwarzaniem zdarzeń od użytkownika, które zosta
lub sekwencje znaków, a następnie wysłane do sterownika terminala. Posługując się
'sterujacy'; read» zaprezentuj działanie znaków sterujących oraz sekwencji «CS
«cat» i sprawdź jak zachowuje się naciśnięcie klawiszy funkcyjnych i kursora. Cz
zachowanie programu «cat» jest inne niż powłoki poleceń?

Zadanie 5

Zadanie 5. Mając na uwadze bieżącą konfigurację sterownika terminala wykonaj n

1. Zaprezentuj edycję wiersza na przykładzie polecenia «cat».
2. Wstrzymaj zadanie pierwszoplanowe «sleep 1000» i przy pomocy wbudowa
   «bg» przenieś to zadanie do wykonania w tle. Jaki sygnał został użyty do wst
3. Uruchom «find /». W trakcie jego działania naciśnij na przemian kilkukrotı
   «CTRL+S» oraz «CTRL+Q». Czemu program zatrzymuje się i wznawia swoją
   terminala nie wysyłał do niego żadnych sygnałów?
4. Uruchom w powłoce «bash» polecenie «cat - &». Czemu zadanie zostało od
   sygnał otrzymało? Zakończ to zdanie wbudowanym poleceniem powłoki «kil
5. Porównaj działanie polecenia «cat /etc/shells &» przed i po zmianie konfi
   niem «stty tostop». Jaki efekt ma włączenie flagi «tostop» na zachowanie

1.

> `cat "jakieś losowe słowa dla testu"` Teraz wykonując ^W usuwamy słowa, ^U usuwa cały wiersz. Możemy również zatrzymać cata, ale wpisywane znaki nadal będą umieszczane w kolejce. Zobaczmy to wykonując napierw ^S, wpisując dowolne znaki a następnie wprowadzając ^Q

2.

> `sleep 1000` `sigtrace [pidof sleep] &` `bg`

3.

> `find /` Wciskając ^S na przemian z ^Q odpowiednio zatrzymujemy i wznawiamy wypisywanie znaków z kolejki na wyjście, warto tutaj zauważyć, że zatrzymujemy tylko kolejkę a nie sam proces (proces nie otrzymuje w ten sposób żadnego sygnału)

4.

> `cat - &` `sigtrace [pidof cat]` Jak widzimy proces zatrzymał się w reakcji na sygnał SIGTTIN. Dzieje się tak ponieważ został uruchomiony w tle, a następnie chciał czytać z wejścia standardowego. Nie może tego zrobić, gdyż tę akcję może wykonać tylko grupa pierwszoplanowa.

5.

```
* [-]tostop
       stop background jobs that try to write to the terminal
```

5. `cat /etc/shells &` wypisze zawartość pliku, po czym nastąpi zakończenie programu. Polecenie stty tostop wysyła sygnał SIGTTOU do procesów drugoplanowych, przez co zostaje dla nich zablokowane wypisywanie do terminala, więc nie zobaczymy nic po ponownym uruchomieniu `cat /etc/shells &`

Zadanie 6

Ściągnij ze strony przedmiotu archiwum «so21_lista_3.tar.gz», następnie rozpakuj i zapoznaj si

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w koment

Zadanie 6. Procedury `setjmp(3)` i `longjmp(3)` z biblioteki standardowej języka C nielokalnych skoków. Uproszczone odpowiedniki tych procedur znajdują się w pliku a definicja «Jmpbuf» w pliku «include/csapp.h». Wyjaśnij co robią te procedury, uczestników zajęć przez ich kod. Dlaczego «Jmpbuf» nie przechowuje wszystkich reje «Longjmp» zapisuje na stos wartość przed wykonaniem instrukcji «ret»?

Ogólny szkic zachowania procedur `setjmp` i `longjmp`

Procedura `setjmp(env)` zapisuje kontekst `env` i zwraca 0. Procedura `longjmp(env, val)` przywraca kontekst `env` "wracając" do `setjmp`. Z punktu widzenia programu wygląda to tak jakby `setjmp` ponownie zwrócił wartość (`val`). `longjmp` nie podejmuje próby zachowania rejestrów *caller-saved* a zatem nie jest to całkowite przywrócenie stanu wyjściowego.

Uproszczona procedura `Setjmp` zapamiętuje obecny kontekst ładując go do structa `Jmpbuf`:

```
typedef struct {
  long rbx;
  long rbp;
  long r12;
  long r13;
  long r14;
  long r15;
  void *rsp;
  void *rip;
} Jmpbuf[1];
```

```
# offsety dla kolejnych pól w Jmpbuf
_JB_RBX = 0
_JB_RBP = 1
_JB_R12 = 2
_JB_R13 = 3
_JB_R14 = 4
_JB_R15 = 5
_JB_RSP = 6
_JB_RIP = 7



        .text

        .globl Setjmp
        .type Setjmp,@function
Setjmp:
    # zapisujemy adres powrotu z Setjmp w %r11, wykorzystamy go później
    # do wykonania skoku
    movq    (%rsp),%r11

    # oczywiście wskaźnik na structa jest przekazany w %rdi
    # przy jego użyciu zapisujemy wymienione niżej rejestry:
    movq    %rbx,(_JB_RBX * 8)(%rdi)
    movq    %rbp,(_JB_RBP * 8)(%rdi)
    movq    %r12,(_JB_R12 * 8)(%rdi)
    movq    %r13,(_JB_R13 * 8)(%rdi)
    movq    %r14,(_JB_R14 * 8)(%rdi)
    movq    %r15,(_JB_R15 * 8)(%rdi)
    movq    %rsp,(_JB_RSP * 8)(%rdi)
    movq    %r11,(_JB_RIP * 8)(%rdi)

    # na koniec zwracamy 0
    xorl %eax,%eax
    ret
        .size Setjmp, . - Setjmp

        .globl Longjmp
        .type Longjmp,@function


Longjmp:

    # przywracamy wartości zapisanych w Jmpbufie rejestrów
    movq    (_JB_RBX * 8)(%rdi),%rbx
    movq    (_JB_RBP * 8)(%rdi),%rbp
    movq    (_JB_R12 * 8)(%rdi),%r12
    movq    (_JB_R13 * 8)(%rdi),%r13
    movq    (_JB_R14 * 8)(%rdi),%r14
    movq    (_JB_R15 * 8)(%rdi),%r15
    movq    (_JB_RSP * 8)(%rdi),%rsp
    movq    (_JB_RIP * 8)(%rdi),%r11 # adres powortu Setjmp

    # w %rsi dostajemy wartość do zwrócenia *val*
    movl %esi,%eax
    testl %eax,%eax

    # jeśli *val* jest zerem to ją inkrementujemy, jak nie to po prostu zwracamy
    jnz 1f
    incl %eax

1: movq %r11,(%rsp) # podmieniamy adres powrotu na powrotu Setjmp
    ret # i wracamy tak jak wróciłby Setjmp
        .size Longjmp, . - Longjmp
```

Dlaczego `Jmpbuf` nie przechowuje wszystkich rejestrów?

Procedura przestrzega konwencji zapisywania tylko rejestrów *colee-saved*. Funkcja wołująca powinna sama zabezpieczyć potrzebne *caller-saved* rejestry.

Zadanie 7

Zadanie 7. Uzupełnij program «game» tj. prostą grę w szybkie obliczanie sumy
procedury «readnum» jest wczytać od użytkownika liczbę. Jeśli w międzyczasie przyjd
ma natychmiast wrócić podając numer sygnału, który przerwał jej działanie. W przeciw
zero i przekazuje wczytaną liczbę przez pamięć pod wskaźnikiem «num_p». Twoja in
«readnum» musi wczytać cały wiersz w jednym kroku. Należy wykorzystać proc
sigsetjmp(3) i alarm(2). Pamiętaj, żeby po wczytaniu ciągu znaków zakończyć go
czasomierz! Kiedy Twój program będzie zachowywać się poprawnie zamień procedur
na longjmp(3) i setjmp(3). Czemu program przestał działać?

```
#include "csapp.h"
#include "terminal.h"

#undef MAXLINE
#define MAXLINE 120

static sigjmp_buf env;

static void signal_handler(int signo) {
  /* TODO: Something is missing here! */
  siglongjmp(env, signo);
```

```
  siglongjmp(env, signo);

  /* Handler blokuje wyzwalający go sygnał przy wywołaniu i odblokowuje go wracając.
     Jeśli użylibyśmy tutaj longjmp to sygnał pozostałby zablokowany. Na zawsze.. */
}

/* If interrupted by signal, returns signal number. Otherwise converts user
 * provided string to number and saves it under num_p and returns zero. */
static int readnum(int *num_p) {
  char line[MAXLINE];
  int n;

  /* TODO: Something is missing here! Use Read() to get line from user. */
  alarm(1);
  n = sigsetjmp(env, 2);
  if (n == SIGALRM || n == SIGINT) return n;

  Read(STDIN_FILENO, line, MAXLINE);
  alarm(0); // jeśli zero to przerywa wszystkie odliczające alarmy

  *num_p = atoi(line);
  return 0;
}

static void game(void) {
  int tty = tty_open();

  int timeout = 0, num1 = 0, num2 = 0, sum;
  int last_sig = 0;
  int lives = 3;
  int score = 0;

  while (lives > 0) {
    switch (last_sig) {
      case 0:
        timeout = 5;
        num1 = random() % 100;
        num2 = random() % 100;
        printf("What is the sum of %d and %d?\n", num1, num2);
        break;

      case SIGINT:
        printf(CHA(1) EL() "Bye bye!\n");
        exit(EXIT_FAILURE);

      case SIGALRM:
        timeout--;
        if (timeout < 0) {
          last_sig = 0;
          lives--;
          printf(CHA(1) EL() "Answer was %d!\n", num1 + num2);
          continue;
        }
        break;

      default:
        app_error("lastsig = %d not handled!\n", last_sig);
        break;
    }

    /* Rewrite user prompt to show current number of lives and timeout. */
    sigset_t set, oldset;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGALRM);
    Sigprocmask(SIG_BLOCK, &set, &oldset);

    int x, y;
    tty_curpos(tty, &x, &y);
    dprintf(STDOUT_FILENO, CHA(1) "lives: %d timeout: %d > ", lives, timeout);
    if (last_sig == SIGALRM)
      dprintf(STDOUT_FILENO, CHA(%d), y);

    Sigprocmask(SIG_SETMASK, &oldset, NULL);

    /* Read a number from user. */
    last_sig = readnum(&sum);
    if (last_sig)
      continue;

    /* Line contains user input (a number) terminated with '\0'. */
    if (sum == num1 + num2) {
      printf("Correct!\n");
      score++;
    } else {
      printf("Incorrect!\n");
      lives--;
    }
  }

  Close(tty);

  printf("Game over! Your score is %d.\n", score);
}

int main(void) {
  /* Initialize PRNG seed. */
  struct timeval tv;
  gettimeofday(&tv, NULL);
  srandom(tv.tv_usec);

  /* SIGALRM is used for timeouts, SIGINT for graceful exit. */
  Signal(SIGALRM, signal_handler);
  Signal(SIGINT, signal_handler);
```

```
  game();

  return EXIT_SUCCESS;
}
```

Zadanie 8

Zadanie 8. Program «coro» wykonuje trzy współprogramy[3] połączone ze sobą w pot
Pierwszy z nich czyta ze standardowego wejścia znaki, kompresuje białe znaki i zlic
wszystkie znaki niebędące literami. Trzeci zmienia wielkość liter i drukuje znaki na s

W wyniku wykonania procedury «coro_yield» współprogram przekazuje niezerow
współprogramu, który otrzyma tę wartość w wyniku powrotu z «coro_yield». Efekty
mentuje **zmianę kontekstu**. Taką prymitywną formę **wielozadaniowości kooperac
multitasking) można zaprogramować za pomocą setjmp(3) i longjmp(3).

Zaprogramuj procedurę «coro_switch» tak, by wybierała następny współprogram d
czała na niego kontekst. Jeśli współprogram przekazał wartość parametru «EOF», to
aktywnych współprogramów.

Program używa listy dwukierunkowej «TAILQ» opisanej w queue(3). Zmienna «r
listę aktywnych współprogramów, «running» bieżąco wykonywany współprogram, a «
programu, do którego należy wrócić, po zakończeniu wykonywania ostatniego aktyw

```
#include "queue.h"
#include "csapp.h"

#define CORO_STKSIZE 4096
#define CORO_STKALIGN 16 /* As required by SysV ABI ! */

#ifndef EOF
#define EOF (-1)
#endif

#ifndef NOTHING
#define NOTHING (-2)
#endif

// super zadanie kocham je, do niezobaczenia
typedef struct coro {
  TAILQ_ENTRY(coro) co_link;
  const char *co_name;
  void *co_stack;
  Jmpbuf co_ctx;
} coro_t;

static TAILQ_HEAD(, coro) runqueue = TAILQ_HEAD_INITIALIZER(runqueue);
static coro_t *running;
static Jmpbuf dispatcher;

/* Initialize coroutine stucture with stack. */
static void coro_init(coro_t *co, const char *name) {
  memset(co, 0, sizeof(coro_t));
  co->co_name = name;
  /* Allocates a fresh stack for the coroutine! */
  if (posix_memalign(&co->co_stack, CORO_STKALIGN, CORO_STKSIZE) < 0)
    unix_error("posix_memalign error");
}

/* Detach a stack from coroutine structure. */
static void coro_destroy(coro_t *co) {
  free(co->co_stack);
}

/*
 * Switch between subsequent coroutines.
 *
 * Dead coroutines, i.e. ones that returned EOF, get removed from the run queue.
 * Feed next coroutine (value returned from coro_yield) with the result from
 * previous one (parameter passed to coro_yield).
 * Return to dispatcher if there're no more coroutines to run.
 */
static noreturn void coro_switch(int v) {
  coro_t *curr = running;
  /* TODO: Use description above to implement the body. */

  if (v == EOF) TAILQ_REMOVE(&runqueue, curr, co_link);

  if (TAILQ_EMPTY(&runqueue)) Longjmp(dispatcher, EOF);

  if ( (running = TAILQ_NEXT(running, co_link)) == NULL )
      running = TAILQ_FIRST(&runqueue);

  Longjmp(running->co_ctx, v);
```

```
}

/* Save caller context and switch back to next coroutine. */
static int coro_yield(int v) {
  int nv = Setjmp(running->co_ctx);
  if (nv == 0)
    coro_switch(v);
  return nv;
}

/* Configure coroutine context to be executed. */
static void coro_add(coro_t *co, void (*fn)(int)) {
  int v = Setjmp(co->co_ctx);
  if (v) {
    /* This will get executed when coroutine is entered first time. */
    fn(v);
    /* Coroutine must pass EOF to be removed from runqueue! */
    coro_switch(EOF);
  }
  /* Coroutine will be running on its private stack! */
  co->co_ctx->rsp = co->co_stack + CORO_STKSIZE;
  TAILQ_INSERT_TAIL(&runqueue, co, co_link);
}

/* Take first coroutine and feed it with passed value. */
static int coro_run(int v) {
  running = TAILQ_FIRST(&runqueue);
  int nv = Setjmp(dispatcher);
  if (nv == 0)
    Longjmp(running->co_ctx, v);
  return nv;
}

/*
 * Actual coroutines that perform some useful work.
 */

static void func_1(int _) {
  int words = 0;
  char prev_ch = ' ';
  char ch;

  while (Read(0, &ch, 1) > 0) {
    if (isspace(ch)) {
      if (isspace(prev_ch))
        continue;
      words++;
    }
    coro_yield(ch);
    prev_ch = ch;
  }

  if (!isspace(ch))
    words++;

  dprintf(STDERR_FILENO, "\nfunc_1: words = %d\n", words);
}

static void func_2(int ch) {
  int removed = 0;

  while (ch != EOF) {
    if (!isalpha(ch)) {
      removed++;
      ch = NOTHING;
    }
    ch = coro_yield(ch);
  }

  dprintf(STDERR_FILENO, "func_2: removed = %d\n", removed);
}

static void func_3(int ch) {
  int printed = 0;

  while (ch != EOF) {
    if (ch != NOTHING) {
      printed++;
      if (islower(ch))
        ch = toupper(ch);
      else if (isupper(ch))
        ch = tolower(ch);
      Write(STDOUT_FILENO, &ch, 1);
    }
    ch = coro_yield(NOTHING);
  }

  dprintf(STDERR_FILENO, "func_3: printed = %d\n", printed);
}

int main(void) {
  coro_t co[3];

  coro_init(&co[0], "func_1");
  coro_init(&co[1], "func_2");
  coro_init(&co[2], "func_3");
  coro_add(&co[0], func_1);
  coro_add(&co[1], func_2);
  coro_add(&co[2], func_3);
  coro_run(NOTHING);
  coro_destroy(&co[0]);
  coro_destroy(&co[1]);
  coro_destroy(&co[2]);

  dprintf(STDERR_FILENO, "Bye-bye!\n");
```

```
  dprintf(STDERR_FILENO, "Bye, bye!\n");

  return EXIT_SUCCESS;
}
```