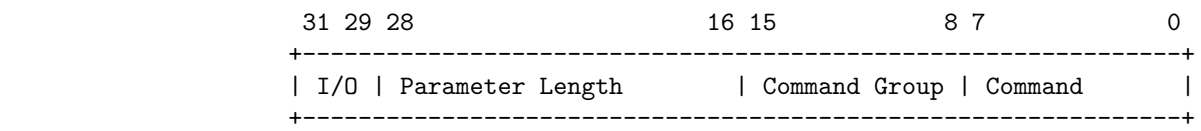
Zadanie 4.

Ioctl - służy do komunikacji z urządzeniami. Ma trzy argumenty:

1. fd urządzenia
2. request code (zależy od urządzenia) i zawiera takie informacje:



- I/O - opis, czy parametry są IN/OUT, a może ich nie ma
- długość parametrów
- polecenie

3. wskaźnik na pamięć, argumenty (ich semantyka zależy od request code)

Semantyka jest zróżnicowana w zależności od urządzenia i requestu. W artykule było napisane, że służy do robienia rzeczy, do których normalny protokuł I/O linuxa jest nieprzystosowany i nie ma na to dobrych rozwiązań.

Te request cody składa się za pomocą mark i definicji odpowiednich poleceń, np.:

```
// Disk-specific ioctls.
#define DIOCEJECT   _IOW('d', 112, int) /* eject removable disk */


#define KIOCTYPE    _IOR('k', 9, int)   /* get keyboard type */

//ioctls for sockets
#define SIOCGIFCONF _IOWR('i', 38, struct ifconf)   /* get ifnet list */
```

Przykłady requestów – dla dysku: odczytaj informację na temat partycji, odczytaj rozmiar sektora.

Recap artukułu z zadania: Jest bardzo dużo wersji tego wywołania systemowego, a często są źle udokumentowane. Do ich obsługi jest dużo definicji i makr. Problem tkwi w słabości Unixowego modelu obsługi urządzeń, który nie umożliwia eleganckiej obsługi dodatkowych operacji.

Btw. polecam komentarze: plik z KIOCTYPE.

Zadanie 5.

(Ta biblioteka (libcsapp) jest dostarczona w paczce do listy.)

CPR – cursor position report

TCGETS - (to samo co *tcgetattr(fd, argp)*) zapisuje obecne ustawienia terminala (np. baud rate - interwał w jakim przesyłane są informacje) w _struct termios

*argp_.

TCSETSW – zmień ustawienia terminala lub serial portu, ale najpierw *"allow output buffer to drain"* (czyli zanim zmieni wysyła to, co już ma wpisane).

TIOCINQ – odczytaj liczbę bitów w buforze

TIOCSTI – wrzuca podany bajt na kolejkę inputu (udawany input)

Flagi ustawień terminala

1. **ECHO** – wpisywane znaki są zwracane do terminala (widoczne na ekranie)
2. **ICANON** – włącza przetwarzanie kanoniczne (*canonical pocessing*), pozwala na to, żeby *ERASE* i *KILL* edytowały wejście i składa je w linie. Jeżeli jest wyzerowana, *read requests* są spełniane bezpośrednio z kolejki wejścia (po czytaniu dostatecznie wielu bajtów lub upływie określonego czasu). Razem z innymi flagami:
   - **ECHOE** – znak *ERASE* powoduje usunięcie ostatniego znaku z linii z ekranu,
   - **ECHOK** – znak *KILL* odrzuca obecną linię, terminal drukuje '\n' po *KILL*,
   - **ECHOKE** – __KILL___ odrzuca obecną linię i usuwa ją z ekranu,
   - **ECHOPRT** – traktuje wyświetlacz jak drukarkę (*"prints a backslash and the erased characters when processing ERASE characters, followed by a forward slash"*).
3. **CREAD** – włącza obieranie znaków. Jeżeli jest wyłączona, terminal nie odbiera żadnego znaku (nie zawsze wspierane przez sprzęt).

Z manuala:

```
//LOCAL MODES:
       ECHOKE      /* visual erase for line kill */
       ECHOE       /* visually erase chars */
       ECHO        /* enable echoing */
       ECHONL      /* echo NL even if ECHO is off */
       ECHOPRT     /* visual erase mode for hardcopy */
       ECHOCTL     /* echo control chars as ^(Char) */
       ICANON      /* canonicalize input lines */
//CONTROL MODES (basic terminal hardware control)
       CSIZE       /* character size mask */
       CS5         /* 5 bits (pseudo) */
       CREAD       /* enable receiver */
```

Zadanie 6.

pipeline – zbiór procesów, które przekazują sobie swoje standardowe wyjścia na wejście.

przekierowanie – nadpisanie fd jakiegoś procesu efektywnie przekierowując jego wyjście do innego pliku bez ingerencji w kod źródłowy.

```
ps -ef | grep zsh | wc -l > cnt
```

Myślę, że tutaj trzeba zrobić podobny obrazek, jak w [APUE] na str. 305.

sh_1 - powłoka do której wpisano komendę: woła **create**(tworzy plik cnt), forkuje *sh_2* i czeka (woła **waitpid**), a potem woła **close**.

sh_2 - tworzy nową grupę (**setpgrp**), woła **pipe** (2 razy), forkuje *sh_3* i *sh_4*, woła **dup2** (przekierowuje wejście z 2. pipe, a wyjście do pliku od ojca), woła exec (tworzy wc).

sh_3 - woła **dup2** - przekierowuje standardowe wyjście do 1. pipe i woła exec (tworzy ps)

sh_4 - woła **dup2** (dwa razy) - odpowiednio przekierowuje wejście z 1., a wyjście do 2.pipe. Woła exec (tworzy grep).

Uzasadnienie:

1. *sh_1* woła **create**, bo potem ktoś musi zawołać **close**, a *sh_2* woła exec i nie będzie miało jak.
2. *sh_2* woła **setpgrp**, żeby wszystkie procesy ps, grep, wc były w jednej grupie innej niż *sh_1*
3. *sh_2* tworzy pipe'y, bo jest rodzicem tych, co się mają komunikować.
4. *sh_2* robi exec dla ostatniego procesu (wc), żeby *sh_1* czekało na wykonanie ostatniego polecenia