Lista 4

tags: SO

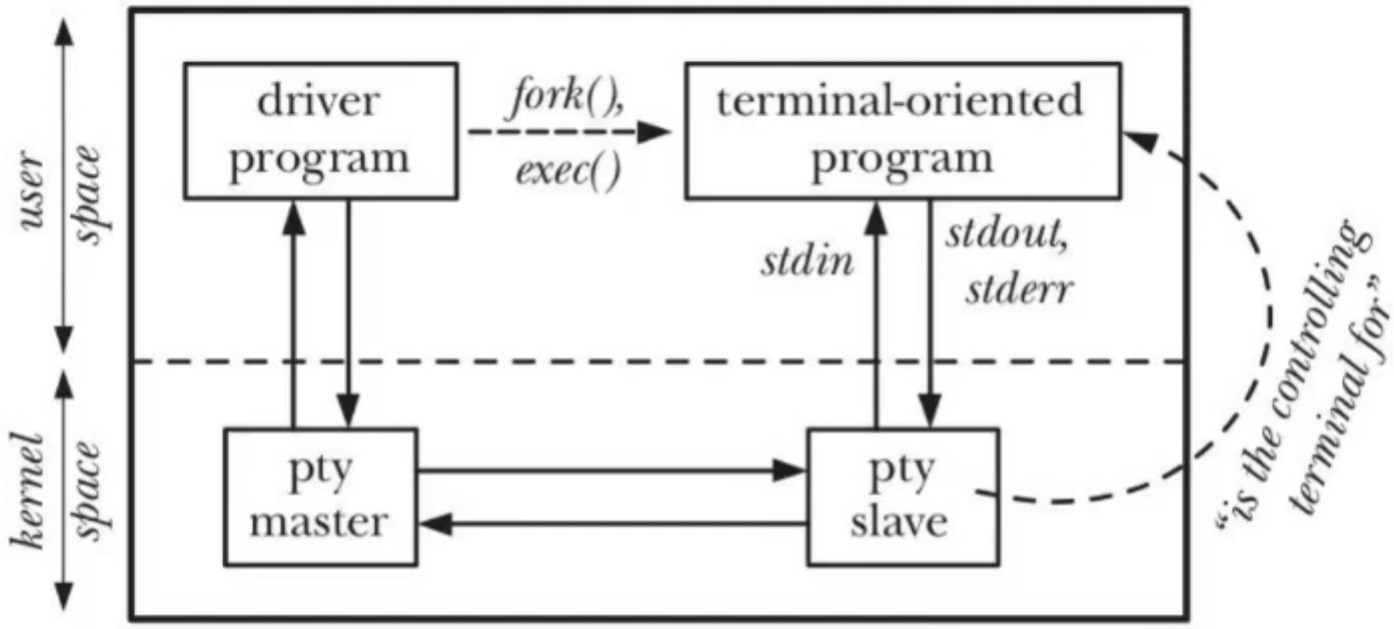


Table 62-1: Terminal special characters

Character	*c_cc* subscript	Description	Default setting	Relevant bit-mask flags	SUSv3
CR	(none)	Carriage return	^M	ICANON, IGNCR, ICRNL, OPOST, OCRNL, ONOCR	•
DISCARD	VDISCARD	Discard output	^O	(not implemented)	
EOF	VEOF	End-of-file	^D	ICANON	•
EOL	VEOL	End-of-line		ICANON	•
EOL2	VEOL2	Alternate end-of-line		ICANON, IEXTEN	
ERASE	VERASE	Erase character	^?	ICANON	•
INTR	VINTR	Interrupt (SIGINT)	^C	ISIG	•
KILL	VKILL	Erase line	^U	ICANON	•
LNEXT	VLNEXT	Literal next	^V	ICANON, IEXTEN	
NL	(none)	Newline	^J	ICANON, INLCR, ECHONL, OPOST, ONLCR, ONLRET	•
QUIT	VQUIT	Quit (SIGQUIT)	^\	ISIG	•
REPRINT	VREPRINT	Reprint input line	^R	ICANON, IEXTEN, ECHO	
START	VSTART	Start output	^Q	IXON, IXOFF	•
STOP	VSTOP	Stop output	^S	IXON, IXOFF	•
SUSP	VSUSP	Suspend (SIGTSTP)	^Z	ISIG	•
WERASE	VWERASE	Erase word	^W	ICANON, IEXTEN	

Noncanonical Mode

Some applications (e.g., *vi* and *less*) need to read characters from the terminal without the user supplying a line delimiter. Noncanonical mode is provided for this purpose. In noncanonical mode (`ICANON` unset), no special input processing is performed. In particular, input is no longer gathered into lines, but is instead available immediately.

Noncanonical mode: Terminal input is not gathered into lines. Programs such as *vi*, *more*, and *less* place the terminal in noncanonical mode so that they can read single characters without the user needing to press the *Enter* key.

Zadanie 1

Zadanie 1. W bieżącej wersji biblioteki «libcsapp» znajdują się pliki «termina Przeczytaj [2, 62.1 i 62.2], a następnie zreferuj działanie procedury «tty_curpos» odcz terminala. Do czego służy kod sterujący «CPR» opisany w CSI sequences[1]? Posiłk wytłumacz semantykę rozkazów «TCGETS» i «TCSETSW», wykorzystywanych odpowiedn i `tcsetattr(3)`, oraz «TIOCINQ» i «TIOCSTI». Na podstawie `termios(4)` wy «ICANON», «CREAD» wpływają na działanie sterownika terminala.

:::warning

- CRT – Cathode Ray Tube (tj. historycznie prymitywne wyświetlacze grafiki xD)
- kody sterujące CSI – kody zarządzające kursorem (przesuwanie itd)
- CPR – kod CSI który zwraca obecną pozycję kursora kodem ESC[n;mR, gdzie n to rząd, a m kolumna :::

Mamy dwie procedury do zarządzania atrybutami terminala:

```
int tcgetattr(int fd , struct termios * termios_p );
int tcsetattr(int fd , int optional_actions , const struct termios * termios_p );
```

:::spoiler **optional_actions**:

- TSCANOW – zmiana jest wykonywana natychmiast
- TSCADRAIN – zmiana wykonywana dopiero po wypisaniu poprzedzającego outputu
- TSCAFLUSH – tak jak TSCADRAIN, ale wyrzucamy wszelki oczekujący do tej pory input :::

Komunikacja odbywa się poprzez structa `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 modes
    cc_t c_line;      // Line discipline (nonstandard)
    cc_t c_cc[NCCS];  // Terminal special characters
    speed_t c_ispeed; // Input speed (nonstandard; unused)
    speed_t c_ospeed; // Output speed (nonstandard; unused)
};
```

:::spoiler

- ECHO – wypisuje wprowadzane znaki
- ICANON – ustawia tryb kanoniczny
- CREAD – uruchamia receiver, jeśli jest wyłączony to żaden znak nie zostanie wczytany :::

```c
void tty_curpos(int fd, int *x, int *y) {
  struct termios ts, ots;

  // pobieramy i zapisujemy structa termios
  tcgetattr(fd, &ts);

  // kopia structa
  memcpy(&ots, &ts, sizeof(struct termios));

  // ustawiamy i zapisujemy flagi
  // (tryb niekanoniczny, bez echa i wyłączone czytanie z wejścia terminala)
  ts.c_lflag &= ~(ECHO | ICANON);
  ts.c_cflag &= ~CREAD;
  tcsetattr(fd, TCSADRAIN, &ts);

  // Sprawdzamy liczbę znaków w kolejce wejściowej
  int m = 0;
  /* TODO: Need to figure out some other way to do it on MacOS / FreeBSD. */
#ifdef LINUX
  ioctl(fd, TIOCINQ, &m); // pozyskuje liczbę bajtów w kolejce
#endif

  // czytamy wszystkie znaki z kolejki
  char discarded[m];
  m = Read(fd, discarded, m);

  // Wypisuje na wyjście pozycję kursora jako \033[row;colR
  Write(fd, CPR(), sizeof(CPR()));

  // Zapisuje w buf wcześniej wypisaną pozycję i kończy dopisując '\0'
  char buf[20];
  int n = Read(fd, buf, 19);
  buf[n] = '\0';

  // Przywraca tryb kanoniczny
  ts.c_lflag |= ICANON;
  tcsetattr(fd, TCSADRAIN, &ts);

  // symulując wejście terminala (TIOCSTI) podajemy kolejno wszystkie wczytane znaki
  for (int i = 0; i < m; i++)
    ioctl(fd, TIOCSTI, discarded + i);

  // przywracamy flagi sprzed wywołania funkcji
  tcsetattr(fd, TCSADRAIN, &ots);

  // odczytuje z bufora zapisaną tam wcześniej pozycję kursora do x, y
  sscanf(buf, "\033[%d;%dR", x, y);
}
```

Semantyka TCGETS, TCSETSW, TIOCINQ, TIOCSTI

```
Get and set terminal attributes
    TCGETS    struct termios *argp
           Equivalent to tcgetattr(fd, argp).

           Get the current serial port settings.

    TCSETS    const struct termios *argp
           Equivalent to tcsetattr(fd, TCSANOW, argp).

           Set the current serial port settings.

    TCSETSW   const struct termios *argp
           Equivalent to tcsetattr(fd, TCSADRAIN, argp).

           Allow the output buffer to drain, and set the current serial port settings.
```

```
Buffer count and flushing
    FIONREAD  int *argp
           Get the number of bytes in the input buffer.

    TIOCINQ   int *argp
           Same as FIONREAD.
```

```
Faking input
    TIOCSTI   const char *argp
           Insert the given byte in the input queue.
```

Zadanie 2

Zadanie 2. Na podstawie [1, 19.2] wyjaśnij działanie programu `script(1)`. Na
z powłoką «dash» przy pomocy polecenia «`script -T timing -c dash`». Wykonaj
powłokę przy pomocy polecenia «`exit 42`», po czym odtwórz sesję przy pomocy pol
`-t timing`». Następnie uruchom polecenie powyższe polecenie przy pomocy «stra
`-o script.log`» i na podstawie zawartości pliku «`script.log`» pokaż jak «script
nala do komunikacji z programami działającymi pod kontrolą powłoki «dash». Pokaż
przepisuje znaki zgodnie z flagami «`ICRNL`» i «`ONLCR`» opisanymi w `termios(4)`.

```
script -T timing -c dash
strace -f -e read,write -o script.log
```

:::success **flagi strace**

- **-f** – śledzi sygnały z procesów utworzonych forkiem
- **-e** – okresla jakie sygnały należy śledzić (wymienione po przecinku)
- **-o** – nazwa pliku wyjściowego (output) :::

::: warning

- NL -- Special character on input and is recognized if the ICANON flag is set. It is the line delimiter `\n` ("new line" or "line feed").
- CR -- Special character on input and is recognized if the ICANON flag is set; it is the `\r` ("carriage return"), 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.
- c_iflag -- mask of input modes (settings)
- c_oflag -- mask of output modes (settings)
- ICRNL -- c_iflag flag, enables mapping CR to NL on input
- ONLCR -- c_iflag flag, maps NL to CR-NL (windows style?) :::

```
The script(1) program that is supplied with most UNIX systems makes a copy in a file
of everything that is input and output during a terminal session. The program does this
by placing itself between the terminal and a new invocation of our login shell.
Figure 19.5 details the interactions involved in the script program. Here, we
specifically show that the script program is normally run from a login shell, which
then waits for script to terminate.
While script is running, everything output by the terminal line discipline above
the PTY slave is copied to the script file (usually called typescript). Since our
keystrokes are normally echoed by that line discipline module, the script file also
contains our input. The script file won't contain any passwords that we enter, however,
since passwords aren't echoed.
```

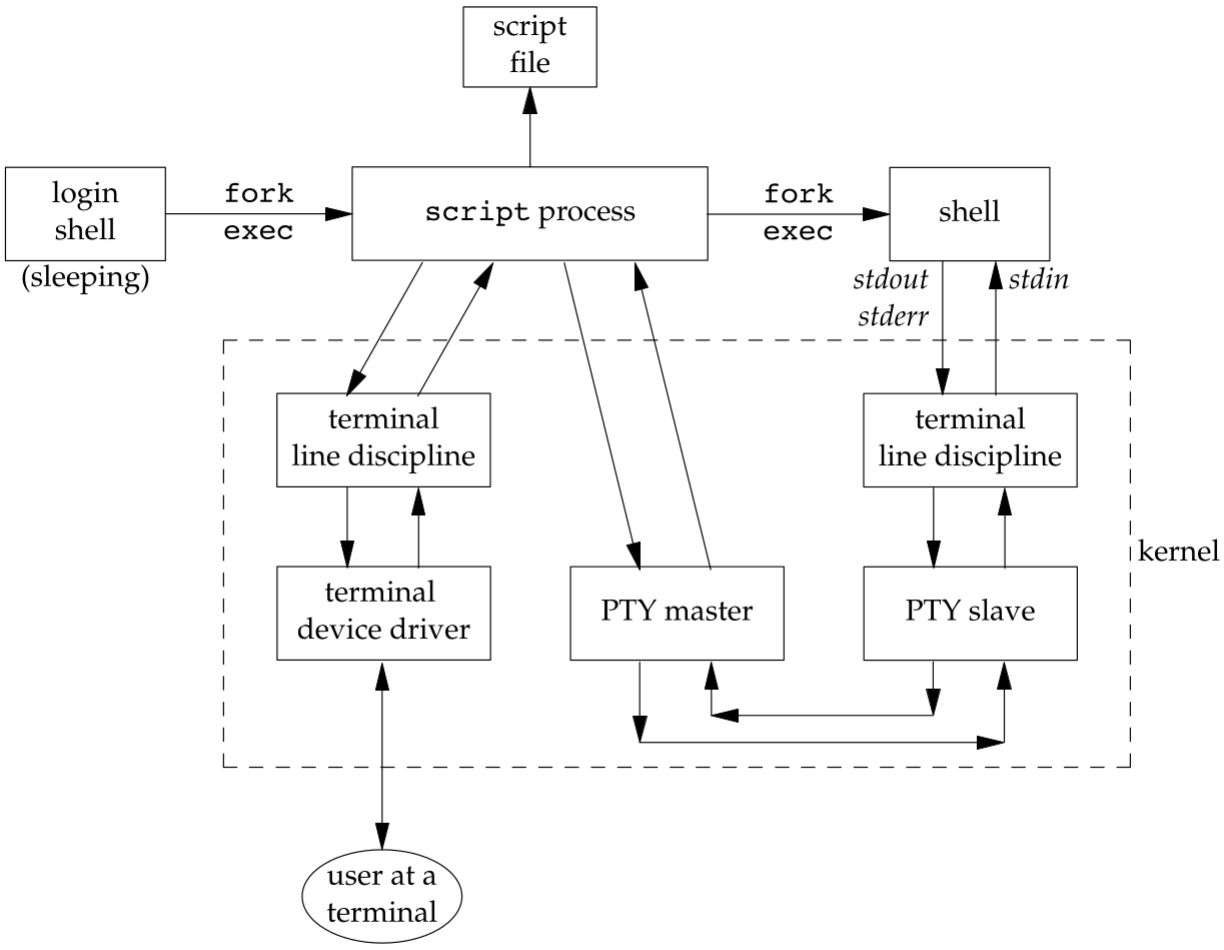


Figure 19.5 The `script` program

Jak script używa pseudoterminala do komunikacji z programami powłoki `dash`?

```
55285 write(2, "$ ", 2)                  = 2
55285 read(0,  <unfinished ...>
55283 read(3, "$ ", 8192)               = 2
55283 write(1, "$ ", 2)                  = 2
55283 read(0, "l", 8192)                = 1
55283 write(3, "l", 1)                  = 1
55283 read(3, "l", 8192)                = 1
55283 write(1, "l", 1)                  = 1
55283 read(0, "s", 8192)                = 1
55283 write(3, "s", 1)                  = 1
55283 read(3, "s", 8192)                = 1
55283 write(1, "s", 1)                  = 1
55283 read(0, "\r", 8192)               = 1
55283 write(3, "\r", 1)                 = 1
```

Można sprawdzić, że "55285" to w tym wypadku `dash`, piszący do scripta "$ ", z kolei "55283" to script. Jak widać mamy włączony tryb "echo", pisząc coś do terminala najpierw puszcza wczytany znak do mastera, a potem od mastera ściąga echo z dasha.

Sterownik terminala czyta znaki zgodnie z flagą ICRNTL

```
57983 read(0,  <unfinished ...>
57981 read(3, "$ ", 8192)
57981 write(1, "$ ", 2)
57981 read(0, "l", 8192)
57981 write(3, "l", 1)
57981 read(3, "l", 8192)
57981 write(1, "l", 1)
57981 read(0, "s", 8192)
57981 write(3, "s", 1)
57981 read(3, "s", 8192)
57981 write(1, "s", 1)
57981 read(0, "\r", 8192)
57981 write(3, "\r", 1)
57983 <... read resumed>"ls\n", 8192)
```

Sterownik terminala pisze znaki zgodnie z flagą ONLCR

```
55344 write(1, "script.log  timing  typescript\n", 31) = 31
55283 read(3, "script.log  timing  typescript\r\n", 8192) = 32
55283 write(1, "script.log  timing  typescript\r\n", 32) = 32
```

Zadanie 3

Zadanie 3. Uruchom **potok** (ang. *pipeline*) «ps -ef | grep sh | wc -l > c
rzonej przy pomocy polecenia «strace -o pipeline.log -f dash». Na pods
«pipeline.log» opisz jak powłoka realizuje funkcje łączenia procesów **rurami**
przekierowanie **standardowego wyjścia** do pliku. W szczególności wskaż które pr
ści będą wołały następujące wywołania systemowe: openat(2) z flagą «O_CREAT
dup2(2), pipe(2), close(2), clone(2) (realizuje fork(2)) i execve(2). Zwróć
kiedy powłoka tworzy rury i kiedy są zamykane ich poszczególne końce.

::: warning **pipe** – creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

dup2(oldfd, newfd) – The dup() system call creates a copy of the file descriptor `oldfd`, using `newfd` for the new descriptor. After a successful return, the old and new file descriptors may be used interchangeably.

openat(dirfd, pathname, flags) – The openat() system call operates in exactly the same way as open(), except for the differences described here. If the pathname given in `pathname` is relative, then it is interpreted relative to the directory referred to by the file descriptor `dirfd` (rather than relative to the current working directory of the calling process, as is done by open() for a relative pathname). Flaga **O_CREAT** sprawia, że jeśli żądany plik nie istnieje to zostaje utworzony. :::

```
strace -e read,write,openat,dup2,pipe,close,clone,execve -o pipeline.log -f dash ps -ef | grep sh | wc -l > cnt
```

Jak widać, pierwszy wiersz to powstanie powłoki `dash`

```
<dash> execve("/usr/bin/dash", ["dash"], 0x7ffe92bed7b8 /* 51 vars */) = 0
```

dash tworzy rurę (4 -> 3) i forkuje w (przyszłego) ps

```
<dash> pipe([3, 4])                     = 0
<dash> clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
```

dash tworzy rurę (4 -> 5) i forkuje w (przyszłego) grep

```
<dash> pipe([4, 5])                     = 0
<dash> clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD <unfinished ...>
```

ps ustawia swoje standardowe wyjście na wejście do rury i robi execve(ps)

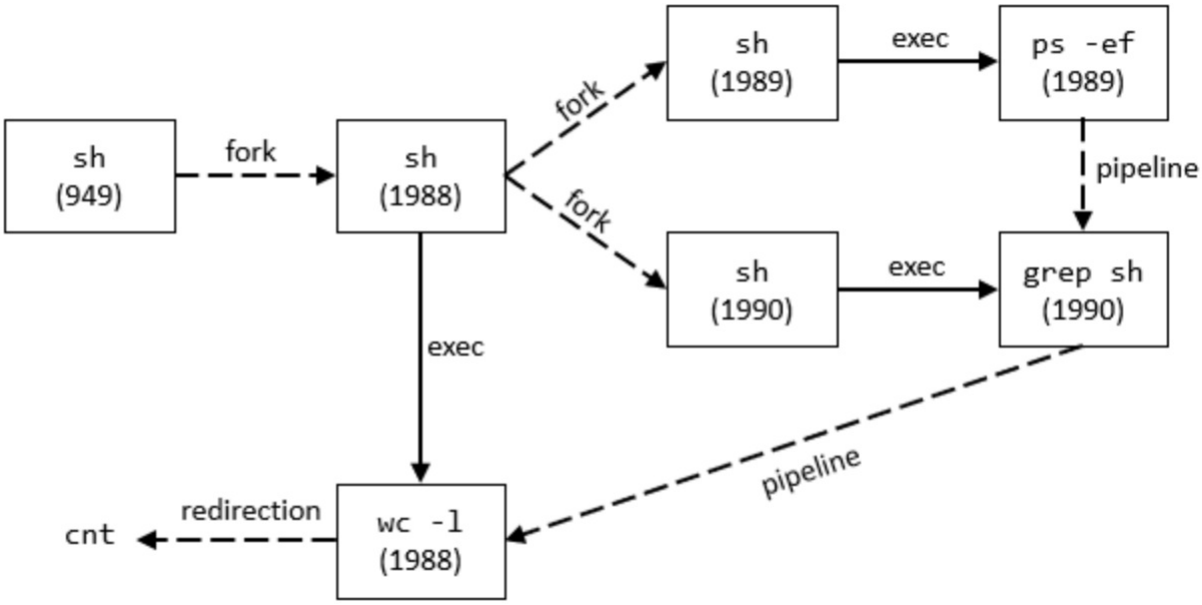
```
<ps> dup(4, 1)
<ps> execve("/usr/bin/ps", ["ps", "-ef"], 0x561c22796d00 /* 51 vars */ <unfinished ...>
```

dash forkuje w przyszłego wc

```
<dash> clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD <unfinished ...
```

itd xD

ogólnie najlepiej pamiętać, że rury są plikami z wejściem i wyjściem i narysować po kolei ten schemat:



Zadanie 4

Zadanie 4. Przyjrzyjmy się raz jeszcze plikowi «`pipeline.log`» z poprzedniego zada
kie procesy należące do potoku muszą zostać umieszczone w jednej grupie procesów
tworzy nową grupę procesów i jak umieszcza tam procesy realizujące potok. Prze
śnij czemu dla każdego podprocesu wywołanie `setpgid(2)` jest robione zarówno
i w procesie potomnym? Kiedy powłoka ustala grupę pierwszoplanową przy pomo
`tcsetpgrp(3)`)? Na jakiej podstawie powłoka wyznacza kod wyjścia potoku?

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns 0 on success, or −1 on error

If *pid* is specified as 0, the calling process's process group ID is changed. If *pgid* is specified as 0, then the process group ID of the process specified by *pid* is made the same as its process ID. Thus, the following *setpgid()* calls are equivalent:

```
setpgid(0, 0);
setpgid(getpid(), 0);
setpgid(getpid(), getpid());
```

If the *pid* and *pgid* arguments specify the same process (i.e., *pgid* is 0 or matches the process ID of the process specified by *pid*), then a new process group is created, and the specified process is made the leader of the new group (i.e., the process group ID of the process is made the same as its process ID). If the two arguments specify different values (i.e., *pgid* is not 0 and doesn't match the process ID of the process specified by *pid*), then *setpgid()* is being used to move a process between process groups.

The typical callers of *setpgid()* (and *setsid()*, described in Section 34.3) are programs such as the shell and *login(1)*. In Section 37.2, we'll see that a program also calls *setsid()* as one of the steps on the way to becoming a daemon.

Several restrictions apply when calling *setpgid()*:

- The *pid* argument may specify only the calling process or one of its children. Violation of this rule results in the error `ESRCH`.
- When moving a process between groups, the calling process and the process specified by *pid* (which may be one and the same), as well as the target process group, must all be part of the same session. Violation of this rule results in the error `EPERM`.
- The *pid* argument may not specify a process that is a session leader. Violation of this rule results in the error `EPERM`.
- A process may not change the process group ID of one of its children after that child has performed an *exec()*. Violation of this rule results in the error `EACCES`. The rationale for this constraint is that it could confuse a program if its process group ID were changed after it had commenced.

Using *setpgid()* in a job-control shell

The restriction that a process may not change the process group ID of one of its children after that child has performed an *exec()* affects the programming of job-control shells, which have the following requirements:

::: warning

- **setpgid(pid, pgid)** – zmienią grupę procesowi **pid** na **pgid**. Jeśli **pid** = 0, to grupę zmienia proces wołający. Jeśli **pid** == **pgid** tworzona jest nowa grupa/
- **ioctl(fd, request, args)** – pozyskiwanie danych z urządzeń (tutaj terminal).
    - TCGETS – request o obecne ustawienia gniazd (?)
    - TIOCSPGRP – ustaw grupę pierwszoplanową tego terminala (podaną w trzecim argumencie) :::

```
strace -e read,write,clone,execve,setpgid,getpgid -o pipeline.log -f dash
```

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

Kiedy powłoka tworzy nową grupę procesów, jak umieszcza tam procesy i dlaczego setpgid wykonywane jest dwa razy?

```
79552 execve("/usr/bin/dash", ["dash"], 0x7ffe3348a958 /* 51 vars */) = 0
79552 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260|\2\0\0\0\0\0"..., 832)
79552 setpgid(0, 79552)                 = 0
79552 write(2, "$ ", 2)                 = 2
79552 read(0, "ps -ef | grep sh | wc -l > cnt\n", 8192) = 31
79552 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, chil
 79560
79552 setpgid(79560, 79560)             = 0
79560 setpgid(0, 79560)                 = 0
79552 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, chil
 79561
79552 setpgid(79561, 79560)             = 0
79561 setpgid(0, 79560)                 = 0
79560 execve("/usr/bin/ps", ["ps", "-ef"], 0x55b188104d10 /* 51 vars */ <unfinished ...>
79552 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, chil
 79562
79552 setpgid(79562, 79560)             = 0
79562 setpgid(0, 79560)                 = 0
79560 <... execve resumed>)             = 0
79561 execve("/usr/bin/grep", ["grep", "sh"], 0x55b188104d30 /* 51 vars */) = 0
```

Jak widać po każdym forku wywoływana jest procedura **setpgid**. Zarówno w rodzicu jak i w dziecku. Zgodnie z dokumentacją **setpgid** można wykonać z poziomu rodzica tylko, jeśli dziecko nie wywołało jeszcze **exec**. Z tego konieczne są dwa wywołania. Procesowi, który już wykonał execve nie da się zmienić grupy z poziomu rodzica, więc nie moglibyśmy tego robić tylko w procesie shella, bo dziecko mogłoby już wykonać execve (nie wiemy w jakiej kolejności się wykonają). Tak samo nie możemy tego zrobić tylko w dziecku, bo nie wiemy ile czasu minie zanim proces dziecka otrzyma czas procesora żeby to zrobić i w międzyczasie mógłby do grupy docelowej zostać wysłany sygnał.

Kiedy powłoka ustala grupę pierwszoplanową?

:::warning `ioctl(fd, TIOCGPGRP, [pid]` ustawia grupę pierwszoplanową na pid :::

```
strace -e read,write,clone,execve,setpgid,getpgid,ioctl -o pipeline.log -f dash
```

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

Po uruchomieniu:

```
80198 execve("/usr/bin/dash", ["dash"], 0x7ffe838bdd18 /* 51 vars
80198 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260
80198 ioctl(0, TCGETS, {B38400 opost isig icanon echo ...}) = 0
80198 ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
80198 ioctl(10, TIOCGPGRP, [80195])     = 0
80198 setpgid(0, 80198)                 = 0
80198 ioctl(10, TIOCSPGRP, [80198])     = 0
```

Po zakończeniu grupy pierwszoplanowej:

```
80198 ioctl(10, TIOCSPGRP, [80198])     = 0
80198 write(2, "$ ", 2)                 = 2
80198 read(0, "exit 42\n", 8192)        = 8
80198 ioctl(10, TIOCSPGRP, [80195])     = 0
80198 setpgid(0, 80195)                 = 0
80198 +++ exited with 42 +++
```

Na jakiej podstawie powłoka wyznacza kod wyjścia potoku?

```
If the reserved word ! does not precede the pipeline, the exit
status is the exit status of the last command specified in the
pipeline.  Otherwise, the exit status is the logical NOT of the
exit status of the last command.  That is, if the last command
returns zero, the exit status is 1; if the last command returns
greater than zero, the exit status is zero.
```

Zadanie 5

Zadanie 5. Czemu nie można czytać i modyfikować katalogów przy pomocy wywołań
Jakim wywołaniem systemowym można wczytać **rekord katalogu** (ang. *directory entr*
katalogu nie jest posortowana? Wyświetl **metadane** katalogu głównego «/» przy po
a następnie wyjaśnij z czego wynika podana liczba **dowiązań** (ang. *hard link*)?

::: info **Rekord katalogu** - struktura zawierająca informacje o plikach w katalogu. Para nazwy pliku i jego ID.

Metadane - informacje o pliku i jego zawartości (nie zawierają nazwy pliku, która jest częścią rekordu, dzięki takiemu rozwiązaniu plik może mieć wiele nazw). Para (Device, inode) unikalnie identyfikuje plik w systemie.

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
}
```

Dowiązanie - jest to referencja wskazująca na istniejącą zawartość pliku lub katalogu umieszczona w tym samym systemie plików. Rozróżniamy dowiązania twarde i symboliczne. Dowiązania twarde to wskaźniki na i-węzły (licznik referencji!) plików różne nazwy tego samego pliku w obrębie jednego systemu plików. Dowiązania symboliczne kodują ścieżkę do której należy przekierować algorytm rozwiązywania nazw. :::

Czemu wywołania *read(2)* i *write(2)* nie działają na katalogach?

Katalog nie jest plikiem z danymi (a read(2) i write(2) operują na plikach), ale z rekordami. read(2) i write(2) wymagają fileops którego katalogi nie mają. **Fileops** - struktura danych opisująca dla danego file descriptora w jaki sposób czytać wejście.

Jakim wywołaniem systemowym można wczytać rekord katalogu(ang.directory entry)?

Kiedyś używało się do tego *readdir(2)*, współcześnie zostało zastąpione przez *getdents(2)*.

Dlaczego zawartość katalogu nie jest posortowana?

Dla wydajności czasowej katalogi w domyśle nie są posortowane (ale np. ext4 są). Można to sprawdzić wykonując `ls -U`.

Wyświetl metadane katalogu głównego «/» przy pomocy polecenia «stat»

Polecenie: `stat /`.

Z czego wynika podana liczba dowiązań (ang. *hard link*)?

TODO

Zadanie 6

Zadanie 6. Intencją autora poniższego kodu było użycie plików jako blokad między
pliku o podanej nazwie w systemie plików oznacza, że blokada została założona. Brak
można założyć. Niestety w poniższym kodzie jest błąd TOCTTOU[2], który opisar
Zlokalizuj w poniższym kodzie wyścig i napraw go! Opowiedz jakie zagrożenia niesie

```
1  #include "csapp.h"
2
3  bool f_lock(const char *path) {
4    if (access(path, F_OK) == 0)
5      return false;
6    (void)Open(path, O_CREAT|O_WRONLY, 0700);
7    return true;
8  }
9
10 void f_unlock(const char *path) {
11   Unlink(path);
12 }
```

Wskazówka: Przeczytaj komentarze do flagi «O_CREAT» w podręczniku do open(2).

::: warning

- **access(pathname, mode)** – access() checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced. The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. **F_OK tests for the existence of the file.** R_OK, W_OK, and X_OK test whether the file exists and grants read, write, and execute permissions, respectively. On success (all requested permissions granted, or mode is F_OK and the file exists), zero is returned.
- **unlink** – unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.
- **O_EXCl** – ensure that this call creates the file: if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() fails with the error EEXIST. This procedure is atomic. :::

Co jest nie tak?

Wykonanie testu `access` i procedury `Open` nie odbywa się atomowo, tj. zewnętrzny proces może wślizgnąć się pomiędzy te dwie instrukcje i utworzyć plik, wtedy funkcja, wbrew naszym intencjom, zwróci `true`. Błąd ten uchodzi pod nazwą TOCTTOU(Time Of Check To Time Of Use) i jest spowodowany założeniem, że między sprawdzeniem stanu pliku, a wykonaniem na nim operacji nie nastąpiła żadna zmiana.

Naprawiony kod

```
#include "csapp.h"

bool f_lock(const char *path) {
    if (Open(path, O_CREAT|O_WRONLY|O_EXCL, 0700) < 0)
        return false;
    return true;
}

void f_unlock(const char *path) {
    Unlink(path);
}
```

Zadanie 7

Ściągnij ze strony przedmiotu archiwum «so21_lista_4.tar.gz», następnie rozpakuj i zapoznaj si

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

Zadanie 7. Program «leaky» symuluje aplikację, która posiada dostęp do danych w torem pliku o nieustalonym numerze kryje się otwarty plik «mypasswd». W wyniku «leaky» uruchamia zewnętrzny program «innocent» dostarczony przez złośliwego

Uzupełnij kod programu «innocent», aby przeszukał otwarte deskryptory plików zawartość otwartych plików do pliku «/tmp/hacker». Zauważ, że pliki zwykłe posi wyjściowego należy wpisać również numer deskryptora pliku i ścieżkę do pliku, tak jak

```
1 File descriptor 826 is '/home/cahir/lista_4/mypasswd' file!
2 cahir:...:0:0:Krystian Baclawski:/home/cahir:/bin/bash
```

Żeby odnaleźć nazwę pliku należy wykorzystać zawartość katalogu «/proc/self/fd» Potrzebujesz odczytać plik docelowy odpowiedniego **dowiązania symbolicznego** przy

Następnie napraw program «leaky» – zakładamy, że nie może on zamknąć pliku Wykorzystaj `fcntl(2)` do ustawienia odpowiedniej flagi deskryptora wymienionej w

Zainstaluj pakiet «john» (John The Ripper[3]). Następnie złam hasło znajdujące się pl niku podatności pozostawionej przez programistę, który nie przeczytał uważnie pod

Wskazówka: Procedura «dprintf» drukuje korzystając z deskryptora pliku, a nie struktury «FILE»

Zadanie 8

Zadanie 8. Uruchom program «mkholes», a następnie odczytaj **metadane** pliku «ho polecenia `stat(1)`. Wszystkie pola struktury «stat» są opisane w `stat(2)`. Oblicz na podstawie liczby używanych bloków «st_blocks» i rozmiaru pojedynczego bloku pliku. Czemu liczba używanych bloków jest mniejsza od tej wynikającej z objętości p Czemu jest większa od liczby faktycznie używanych bloków zgłaszanych przez «m zjawisko na podstawie [1, 3.6].