Informacje o procesach czyli polecenie "ps".

Konsola (xterm)

```
$ tty01
tty01
$ bash
$ sleep 100
```

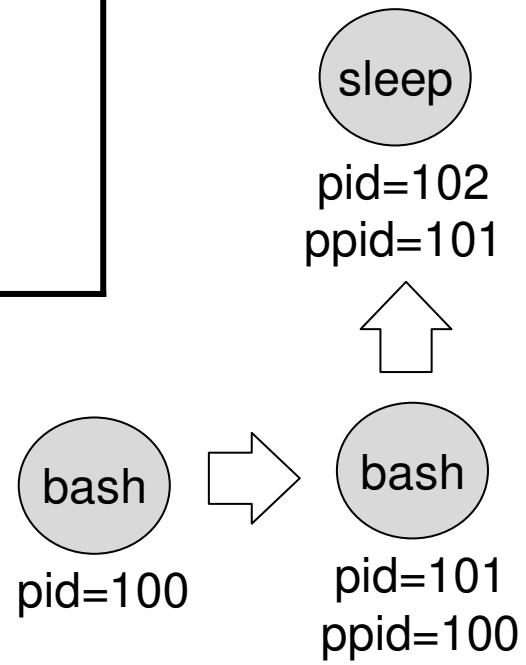
Konsola (xterm)

```
$ ps -t tty01 -o pid,ppid,cmd
PID PPID CMD
100 99   bash
101 100  bash
102 101  sleep 100
```

```
ps -t tty01 -o pid,ppid,cmd
```

które procesy

jakie informacje pokazać



Polecenie "ps" c.d.

```
ps -u mhanckow -o pid,ppid,tty,state,user
ps -t tty01    -o pid,ppid,cmd
```

-o	nazwa	opis
pid	PID	Process IDentifier
ppid	PPID	Parent Process ID (PID procesu macierzystego)
user	USER	(Efektywny) użytkownik procesu
ruser	RUSER	Rzeczywisty użytkownik procesu
uid	UID	User ID
ruid	RUID	Real user ID (real =rzeczywisty)
tname	TTY	Plik specjalny terminala (tzw *terminala sterującego* procesu)
state	S	Stan procesu: R=Run, S=Sleep, I, T
pgid	PGID	ID grupy procesów
pcpu	%CPU	Zużycie czasu procesora

/etc/passwd

Sygnały w Unix-ie.

- sygnały niosą jeden bit informacji ...
- informują proces o jakimś zdarzeniu
  (np o naciśnięciu klawisza Ctrl+C na *terminalu sterującym* procesu)
- są wysyłane do procesu przez:
  - przez system operacyjny (np po naciśnięciu Ctrl+C)
  - przez inny proces (np komenda "kill")
- znaczenie niektórych sygnałów:

SIGINT	2	Wysyłany do procesu po naciśnięciu Ctrl+C; powoduje ***zakończenie procesu***; jednak proces może go *przechwycić* lub *ignorować* ...
SIGKILL	9	***Bezwarunkowe zakończenie procesu*** ; sposób na zakończenie procesu którego nie można inaczej zakończyć ...
SIGTSTP	?	Wysyłany do procesu po naciśnięciu Ctrl+Z; powoduje ***zatrzymanie procesu***, można go potem wznowić (tzw mechanizm *sterowania pracami*)

Sygnały c.d.

- informacje o innych sygnałach:
  ```
  man 7 signal
  ```
- polecenie "kill" służące do wysyłania sygnałów:
  kill -nr_sygnału pid_procesu
  przykłady:
  ```
  kill -9 1234
  kill -kill 1234
  ```
- sygnały mają zdefiniowaną *standardową rekację*
  (zazwyczaj jest to zakończenie procesu; patrz man 7 signal)
- proces może *przechwycić* niektóre sygnały (np SIGINT),
  tj zdefiniować własną procedurę obsługi sygnału
- proces może *ignorować* niektóre sygnały
  (ale SIGKILL nie można ani przechwytywać ani ignorować, zawsze jest obsługiwany standardowo tj powoduje zakończenie procesu)
- *UWAGA !!!* klawisz Ctrl+D powoduje że terminal zachowuje się jak plik który się skończył; nie powoduje wysłania żadnego sygnału (w przeciwieństwie do Ctrl+C, Ctrl+Z) !; przykład:
  ```
  cat >plik.txt
  ```

Deskryptory plików.

- ***deskryptor*** to liczba całkowita którą otrzymujemy po otwarciu pliku
- cykl przetwarzania pliku:
  - `desk=open("plik.txt", ...) ;`
    // otwarcie pliku = przygotowanie pliku do przetwarzania
    // "desk" jest zmienna typu całkowitego
    // open(), read(), write(), close() to funkcje systemowe Unixa (język C)
  - `read(desk, &zm, ...); write(desk, &zm, ...);`
    // operacje czytania i pisania na pliku (za pośrednictwem desk)
  - `close(desk);`
    // zamknięcie pliku (desk przestaje być "ważny")
- deskryptory standardowe:
  - stdin, standardowe wejście, 0
  - stdout, standardowe wyjście, 1
  - stderr, stand. wyjście błędów, 2
- każdy proces po uruchomieniu otrzymuje (zazwyczaj) desk. 0,1,2 związane z jego terminalem ster. (czyli z plikiem typu /dev/tty01 lub /dev/tty)
- co to znaczy że proces "pisze na stdout" ? odp: pisze do desk. 1 !

Przeadresowanie.

- ***przeadresowanie*** to zmiana znaczenia deskryptorów 0,1,2
  przykład:

```
cat plik.txt >plik2.txt
```

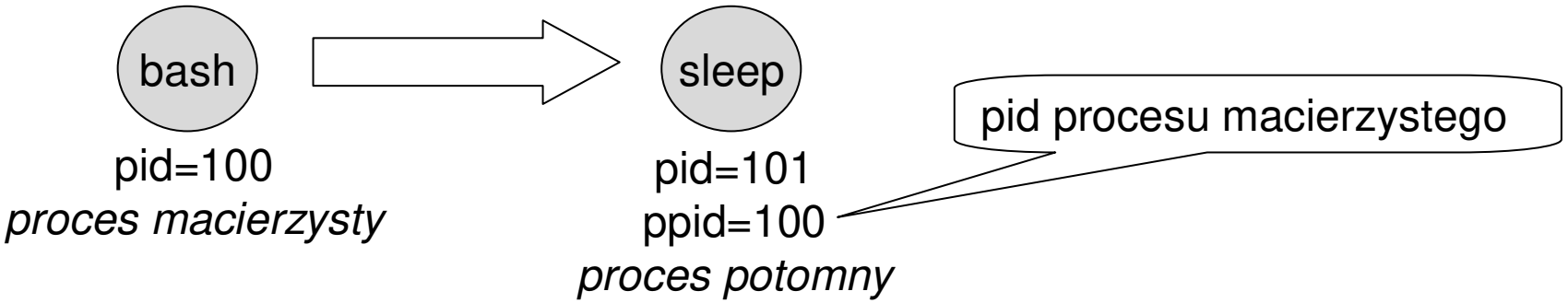
  efekt:

  program "cat" zamiast pisać na terminal będzie pisał do plik2.txt

- jak się realizuje "przeadresowanie" ? (w powyższym przykładzie)
  - wymusza się aby deskryptor 1 odnosił się do plik2.txt (standardowo odnosi się do terminala)
  - program "cat" o tym w ogóle nie wie i pisze do deskryptora 1 sądząc że to terminal !.

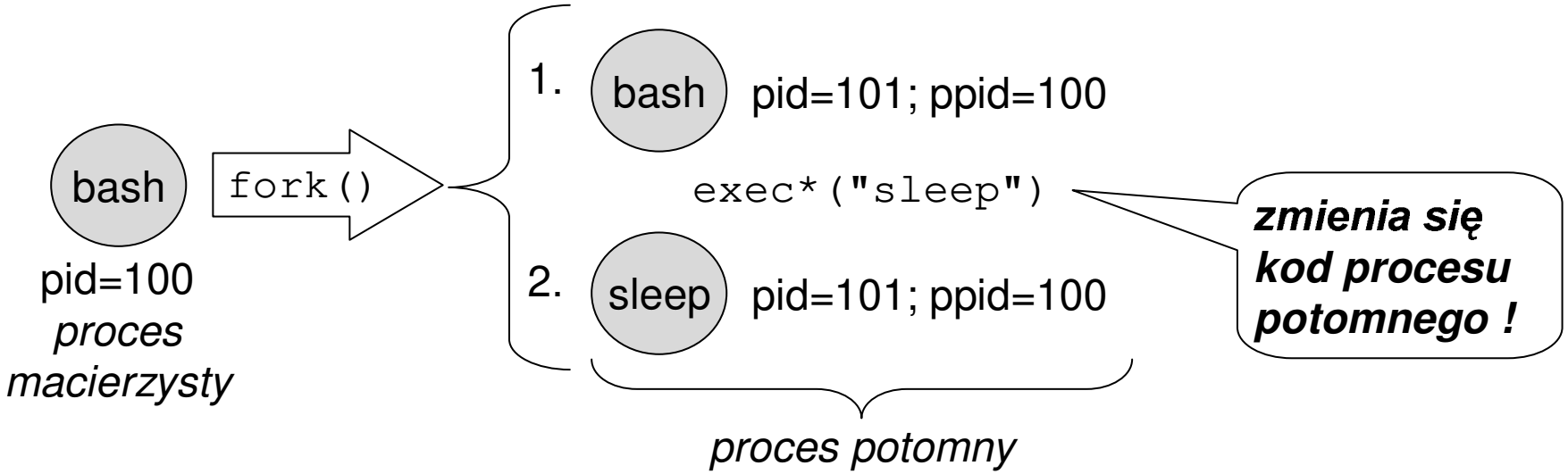
Proces macierzysty i potomny.

- w Unix-ie proces może się "rozdwoić" (funkcja sys. `fork()`)
- nowy proces nazywamy ***procesem potomnym***, a stary proces nazywamy ***procesem macierzystym***
- proces potomny jest początkowo dokładną kopią procesu macierzystego ("dziedziczy" po nim wiele rzeczy, np. kod programu i otwarte pliki)
- proces potomny może zacząć wykonywać inny program niż ten wykonywany przez proces macierzysty (funkcja sys. `exec*()`)
- jeśli z pewnej powłoki uruchamiamy jakiś program to program ten wykonuje się w procesie potomnym a powłoka jest jego procesem macierzystym
- przykład: jeśli w bash-u wydaliśmy polecenie "sleep 10" to będziemy mieli taką sytuację :

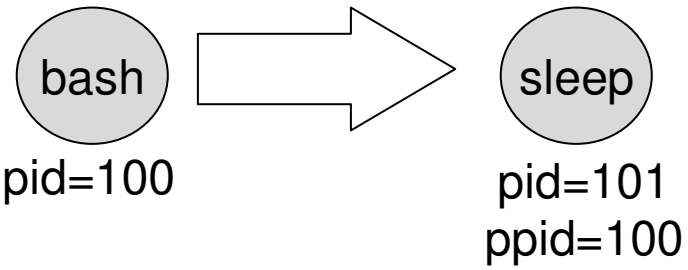


Proces macierzysty i potomny c.d.

- na czym polega uruchamianie programu w Unix-ie (z bash-a uruchamiamy program "sleep 10" ...)



- ... i ostatecznie mamy taką sytuację:



Proces macierzysty i potomny c.d.

- co proces potomny dziedziczy po macierzystym:

`fork()`

`exec*()`

- otwarte pliki
- zmienne środowiska
- bieżący katalog
- kod programu (`exec*(kod_programu)`)
- obsługę sygnałów (?)

•ignorowanie
•standardowa reakcja
•przechwytywanie (?)

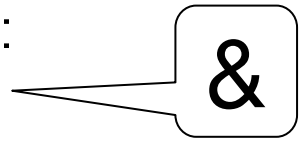
Procesy pierwszo- i drugo- planowe.

- ***proces pierwszoplanowy*** - to taki na którego zakończenie powłoka ***czeka***, zanim pozwoli uruchomić następny program ...
  przykład:

```
sleep 10
  # następny program można będzie uruchomić dopiero
  # gdy ten się skończy ...
```

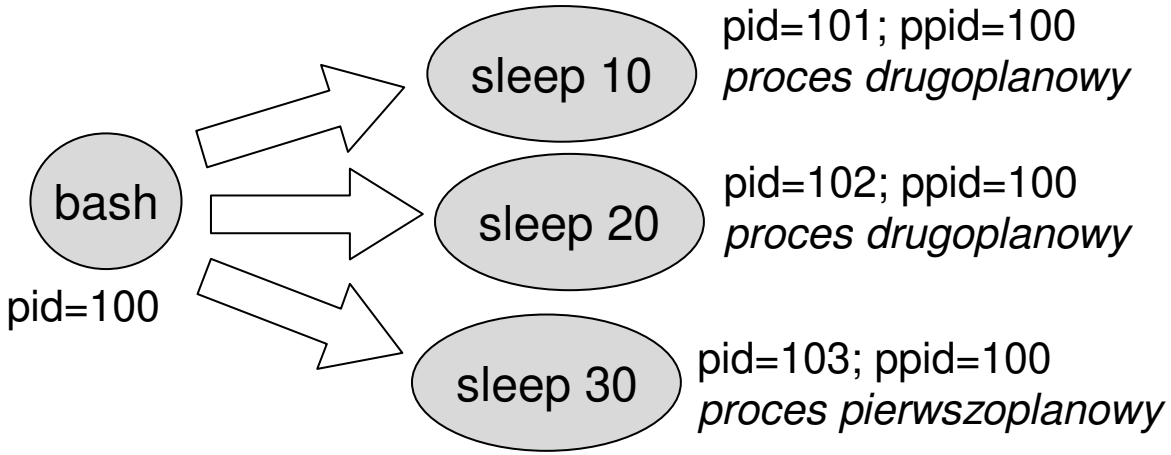
- ***proces drugoplanowy*** - to taki na którego zakończenie powłoka ***nie czeka***; przykład:

```
sleep 10 &
```



- przykład: po wydaniu poniższych poleceń otrzymamy ...

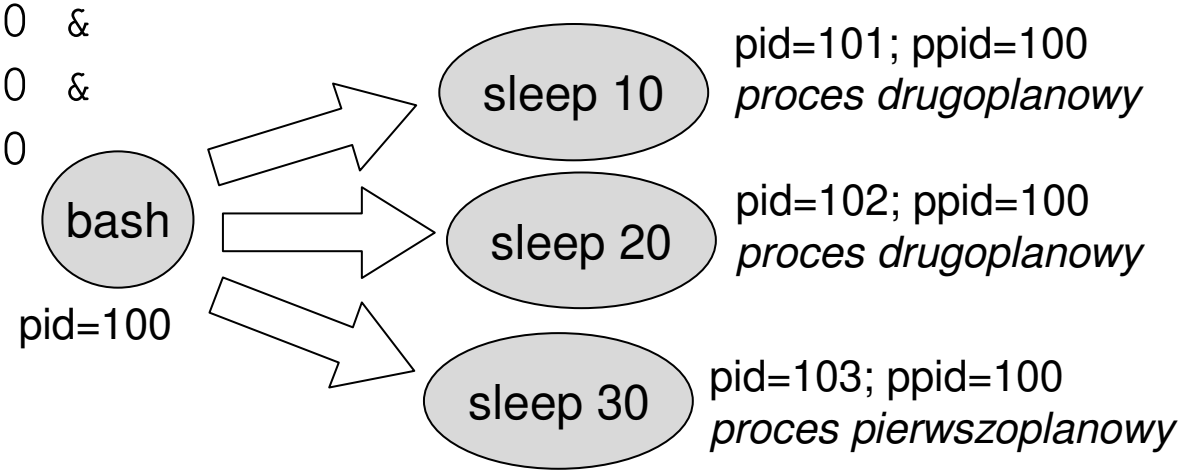
```
sleep 10 &
sleep 20 &
sleep 30
```



Procesy pierwszo- i drugo- planowe c.d.

- przykład: po wydaniu poniższych poleceń otrzymamy ...

```
sleep 10 &
sleep 20 &
sleep 30
```



- *Uwaga !!!*: klawisze Ctrl+C i Ctrl+Z powodują wysłanie odpowiednich sygnałów tylko do procesów pierwszoplanowych !. *Wniosek*: aby zakończyć proces drugoplanowy musimy użyć poleceń: "ps" i "kill" ...

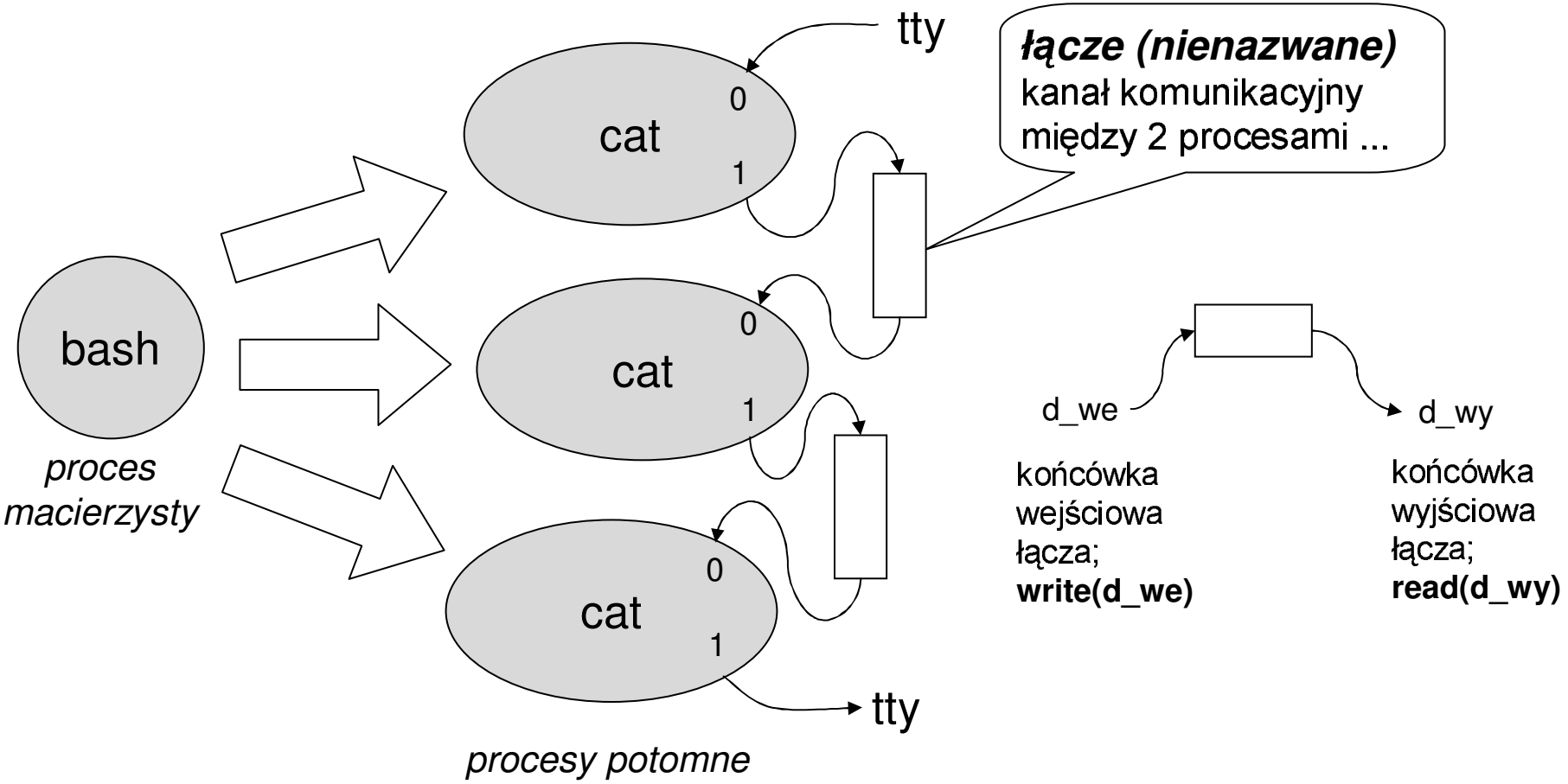
Potoki i łącza.

- przykład potoku:

```
cat | cat | cat
```

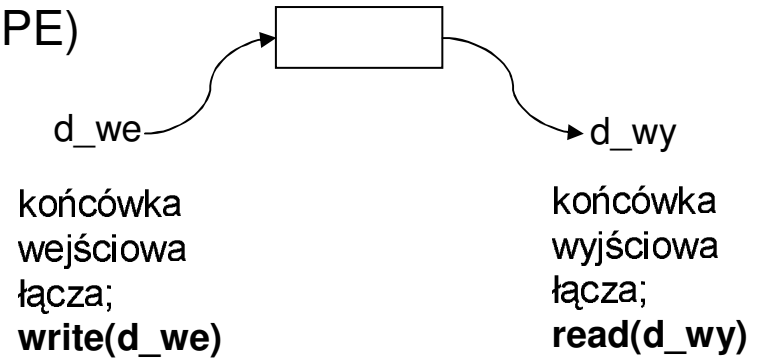
potok

po uruchomieniu powyższego potoku ...



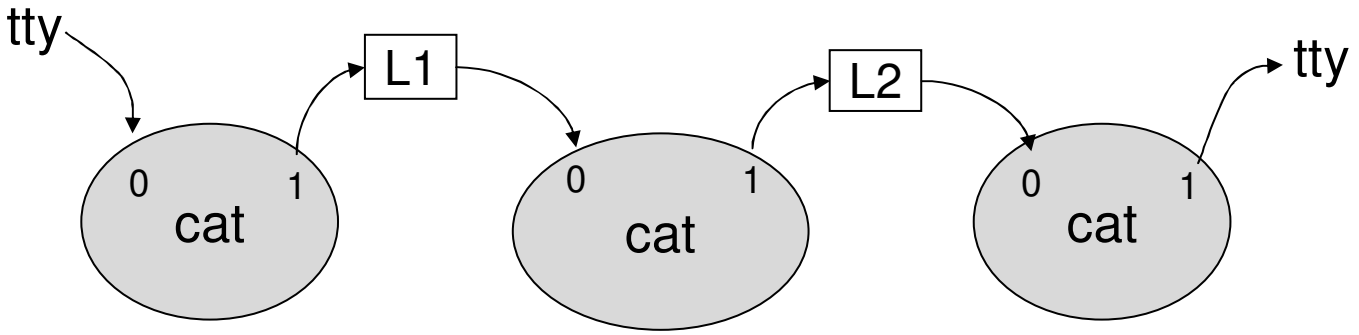
Potoki i łącza c.d.

- potoki DOS-owe są imitacją potoków Unix-owych !
  dowód: wystarczy uruchomić coś podobnego do tego ...
  `cat | cat | cat`
  (kiedy zobaczymy wyniki działania potoku pod DOS-em ?)
- ***prawa rządzące łączami:***
  - mają ograniczoną pojemność !; proces który pisze do "przepełnionego" łącza zostanie uśpiony (aż inny proces nie odczyta danych z łącza)
  - proces czytający z pustego łącza zostanie uśpiony (aż inny proces nie wpisze czegoś do łącza)
  - jeśli czytamy z łącza którego końcówka we nie jest otwarta w żadnym procesie to funkcja `read(d_wy)` zwraca 0 (zachowuje się jakbyśmy czytali sekwencyjnie z pliku i ten plik się skończył ...)
  - podobna zasada przy zapisie (SIGPIPE)



Potoki i łącza c.d.

- mechanizm *automatycznego kończenia się potoku*:
  `cat | cat | cat`
  wpisujemy tekst,
  na terminalu widzimy "echo" wprowadzanych linii ...
  ale dlaczego po naciśnięciu Ctrl+D wszystkie procesy potoku się kończą ?
- potrzebne fakty:
  - program "cat" czyta z stdin i zapisuje to co odczytał na stdout; gdy napotka koniec pliku kończy swoje działanie
  - Ctrl+D naciśnięte na terminalu powoduje że terminal zachowuje się jak plik który się skończył
  - jeśli czytamy z łącza którego końcówka we nie jest otwarta w żadnym procesie to funkcja `read(d_wy)` zwraca 0 (zachowuje się jakbyśmy czytali sekwencyjnie z pliku i ten plik się skończył ...)



Skrypty powłokowe

- Powłoki w Unixie:

  sh, csh, ksh (← tą omówimy), bash

- Główne zadania powłoki:
  - uruchamianie programów przy pomocy komend
    - ułatwienia we wprowadzaniu złożonych komend (TAB [bash], *.txt)
    - konstrukcje zmieniające działanie programu (>, >>, <, &)

    ❖ *konstrukcje pozwalające z istniejących "małych" programów utworzyć nowy "duży" program, np.:*

    ls ⊕ grep = ls | grep "txt"

    pod Unix-em bardzo się do tego zachęca !!!
  - wykonywanie skryptów
    - skrypt to rodzaj programu napisanego w "języku skryptowym" znacząco się różniącym np. od Pascala ...
    - skrypt to plik zawierający komendy i "instrukcje sterujące" (np. if, while)
    - skryptów nie trzeba kompilować !
    - instrukcji sterujących można używać także bezpośrednio z konsoli

Skrypty powłokowe c.d.

- Tworzenie i uruchamianie skryptów (metoda uniwersalna)
  - tworzymy plik ze skryptem, np "skrypt1"
  - pierwsza linia skryptu musi być specjalna (zawiera ścieżkę do interpretera skryptu = powłoki):

```
#!/bin/ksh
# poniżej kod skryptu ...
echo "to ja, Twój skrypt !"
echo "podałeś jako parametry: $1, $2, $3"
i=1
while [[ $i -le 5 ]]
do
  echo ">>>"$i"<<<"; i=$((i+1));
done
echo "Twój skrypt skończył działanie!"
```

  - nadajemy skryptowi prawo "x":

```
chmod u+x skrypt1
```

  - teraz skrypt można uruchamiać jak każdy program ! :

```
skrypt1
```

Konstrukcje języka skryptowego ksh

Opis konstrukcji powłoki
oraz programu "awk"
patrz:

http://main2.amu.edu.pl/~mhanckow

***** SOP121 – ćwiczenia *****

***** Temat C *****

Przykład zastosowania powłoki "ksh" i programu "awk".

- używając następujących "cegiełek" :

  `echo, ps, awk, ksh`

  napiszemy skrypt wypisujący PIDy procesów pochodzących bezpośrednio od procesu o PIDzie podanym przez parametr:

```
#!/bin/ksh
for p in $(ps -o pid,ppid | awk "$1==\$2 {print \$1}")
do
  echo "$p pochodzi od $1"
done
```

- wyjaśnić rolę `$1` i `\$2` w `"$1==\$2 {print \$1}"`