# Lab1 – instalacja debiana

# Leb2 – podstawy basha

jeśli przy przypisywaniu tekstu do zmiennej użyjemy podwójnego cudzysłowiu – wtedy znaki specjalne działają. Jeśli użyjemy pojedynczego – każdy znak jest traktowany jak zwyczajny tekst.

$@ - lista parametrów rozdzielonych spacjami  
$\* - traktuje wszystkie argumenty jako ciąg znaków.

przypisywanie komendy do zmiennej najlepiej robić X=$(ls) ale można robić X=’ls’ (ale ten drugi czasem nie działa)

* cw2 – odwoływanie się do zmiennych (kiedy bierzemy coś w pojedynczy, podwójny cudzysłów, a kiedy wcale nie weźmiemy w cudzysłów)
* cw3 – opis zmiennych path. random, pwd, ps1, user, hostname, ostype
* cw4 – przypisanie ls -l do zmiennej, żeby nie było żadnej różnicy w wyświetleniu
* cw5a – Imię ma kota, kot ma Imię – robimy tak, żeby działało dla dowolnych imion żeńskich i męskich (wyrażenia regularne)
* cw5b – skrypt wczytuje plik jako argument, wypisuje pierwszy i ostatni znak z pliku i zmienia kolor wszystkich wystąpień SOP na zielono (echo -e)
* zaddom – jako argument przyjmuje plik i podświetla wszystkie wystąpienia mojego imienia na czerwono

# Lab3 – bash trochę trduniejsze

pętle, instrukcje warunkowe, funkcje

* cw1\_1 – skrypt sprawdza czy podano więcej niż jeden parametr
* cw1\_2 – sprawdza czy plik znajduje się w bieżącym katalogu. Jeśli tak, wykonuje go, jeśli nie, to niech sprawdzi czy podano argument i wywoła plik z parametru. W innym wypadku komunikat o błędzie
* cw1\_3 skrypt sprawdza czy kończy się .sh, jeśli nie to zmienia nazwę dodając rozszerzenie
* cw1\_4 Skrypt sprawdza czy w bieżącym katalogu jest więcej niż 5 plików, jeśli tak to wypisuje informację że tak jest
* cw2\_1 – skrypt wypisuje liczby od 1 do 10
* cw2\_2 skrypt wypisuje argumenty (podane w cudzysłowach)
* cw2\_3 skrypt wypisuje listę plików i katalogów poprzedzając każdy napisem.
* cw2\_4 skrypt dla każdego pliku z rozszerzeniem .c wywołuję komendę
* cw2\_5 skrypt wczytuje listę plików z wiersza poleceń i wypisuje ich zawartość od początku do końca i od końca do początku
* cw3\_1 skrypt przyjmuje argumenty tekst i kolor – ma pokolorować tekst na zadany kolor
* cw3\_2 funkcja fibbonacciego liczona rekurencyjnie
* zaddom\_1 – skrypt sprawdza czy tekst jest poprawnie napisany polskim (od wielkiej litery, zawiera tylko litery i polskie znaki). Sprawdza czy tekst jest kodem pocztowym, czyli xx-xxx (gdzie x to cyfry) oraz skrypt, który sprawdza czy tekst jest adresem e-mail. Końcowo skrypt rozróżnia imię, kod pocztowy i mail.

# Lab4 wstęp C

struktury, podstawowe komendy, kompilacja. scanf, printf

wskaźniki (również do funkcji)

* zad1\_1 – program obliczający sumę wprowadzonych liczb (ze standardowego wejścia) (scanf, printf)
* zad1\_2 – zad1\_1 tylko suma zwracana jako kod zakończenia programu (i wyświetlamy później poleceniem $?
* zad1\_3 program wyświetlający choinkę o zadanej wysokości (atoi, argc, argv)
* zadIO\_1 – program wczytujący ciąg liczb, pierwsza liczba to ilość podanych liczb, następne to kolejne liczby. program wypisuje je od tyłu. Jeśli podamy jako argument plik, to wpisuje z pliku a jeśli nie to ze standardowego wyjścia (fopen, fclose, fscanf, malloc, calloc, free)
* zadIO\_2 program który wyświetla plik tak jak heksedytor
* zadwskaznik – trzy funkcje ze wskaźnikami przechowywanymi w tablicy. W zależności od podanych argumentów program wybiera odpowiednią funkcję
* zaddom\_1 – lista jednostronnie wiązana
* zaddom\_2 – lista dwustronnie wiązana
* zaddom\_3 – fibbonacci w wersji rekurencyjnej i iteracyjnej

# Lab5 Procesy

fork – tworzy kopię aktualnego procesu (proces dziecko)

* zad1 – program, który uruchomi się, poda swój pid (numer procesu) i ppid (numer procesu rodzica)
* zad2 – program w którym będzie komenda fork(). Proces rodzic powinien poczekać na zakończenie procesu dziecka, oba powinny wyświetlać swój pid oraz ppid
* zad3 – program, który będzie pobierał jeden argument – proces rodzica będzie wyliczał sumę od 0 do podanego argumentu a proces dziecka wypisze wszystkie liczby nieparzyste od 1 do argumentu. Najpierw wypisze proces dziecka a później rodzica.
* zad4 – program stworzy 100 procesów potomnych tak, że każdy kolejny będzie dzieckiem poprzedniego (1 rodzic ma jedno dziecko)
* zad5 – program stworzy 100 procesów potomnych tak, że każdy kolejny będzie dzieckiem pierwszego (1 rodzic ma 100 dzieci)
* zaddom – tworzy drzewo rodzic, 2 dzieci i każde dziecko ma 2 dzieci. Rysuje drzewo procesów.

# Lab6 Sygnały i sockety

sygnały służą do przesyłania jakiejś informacji do procesów, np. o błędnie wprowadzonych argumentach. Można sygnały obsłużyć, czyli napisać kod, który odpowiednio zachowa się po otrzymaniu danego sygnału.

#include <signal.h>

// volatile sig\_atomic\_t flag = 0;

// to jest mniej

void handler (int signum) {

... obsluga sygnalu

}

int main (void) {

struct sigaction new\_action, old\_action;

new\_action.sa\_handler = handler;

sigemptyset (&new\_action.sa\_mask);

new\_action.sa\_flags = 0;

sigaction (SIGINT, NULL, &old\_action);

if (old\_action.sa\_handler != SIG\_IGN)

sigaction (SIGINT, &new\_action, NULL);

sigaction (SIGHUP, NULL, &old\_action);

if (old\_action.sa\_handler != SIG\_IGN)

sigaction (SIGHUP, &new\_action, NULL);

sigaction (SIGTERM, NULL, &old\_action);

if (old\_action.sa\_handler != SIG\_IGN)

sigaction (SIGTERM, &new\_action, NULL);

}

Sigaction ma 3 pola:

* sa\_handler – wskaźnik na funkcję obsługującą sygnał
* sa\_mask – jeżeli jakiś sygnał będzie zamaskowany, to znaczy że będziemy dany sygnał ignorować (ustawia się to poleceniem sigemptyset – nic nie jest zamaskowane, sigfullset – wszystko jest zamaskowane, sigaddset – możemy pojedyncze sygnały dodawać do zamaskowania)
* sa\_flags = 0 (zawsze będzie u nas 0)

jak już to mamy to wywołujemy sigaction(SygnałKtóryChcemyNadpisać, jakąObsługą, NULL)

na zajęciach był oto tworzone tak:

sigset\_t iset;

struct sigaction act;

sigmeptyset(&iset);

act.sa\_handler = &FunkcjaObsługująca;

act.sa\_mask = iset;

act.sa\_flags = 0;

sigaction(sygnał, &act, NULL);

funkcja kill w C przyjmuje najpierw numer PID a później sygnał – kill(PID, sygnał);