2 Pliki

Plik jest pojęciem, z którym spotyka się niemal każdy użytkownik systemu komputerowego, nawet użytkownik końcowy, zajmujący się obsługą komputera w elementarnym zakresie. Popularność tego pojęcia wynika z faktu, że plik jest podstawową formą przechowywania i udostępniania informacji. Dla użytkownika końcowego plik jest zatem abstrakcyjnym obrazem informacji w systemie komputerowym.

Z punktu widzenia systemu operacyjnego lub działających w nim aplikacji plik jest zbiorem odpowiednio powiązanych ze sobą danych, umieszczonych najczęściej w pamięci nieulotnej. Z plikiem związane są pewne atrybuty, z których najważniejszą rolę pełnią identyfikatory, pozwalające w sposób jednoznaczny wskazać konkretny plik w systemie (np. nazwa pliku).

2.1 Operacje na plikach zwykłych

Jądro systemu operacyjnego UNIX udostępnia dwie podstawowe operacje na plikach — *odczyt* i *zapis* — realizowane odpowiednio przez funkcje systemowe `read` i `write`. Z punktu widzenie jądra w systemie UNIX plik nie ma żadnej struktury, tzn. nie jest podzielony na przykład na rekordy. **Plik jest traktowany jako tablica bajtów**, zatem operacje odczytu lub zapisu mogą dotyczyć dowolnego fragmentu pliku, określonego z dokładnością do bajtów.

2.1.1 Otwieranie pliku

Wykonanie operacji wymaga wskazania pliku, na którym operacja ma zostać wykonana. Plik w systemie UNIX identyfikowany jest przez nazwę (w szczególności podaną w postaci ścieżki katalogowej), przy czym podawanie nazwy pliku przy każdym odwołaniu do niego wymagałoby każdorazowego przeszukiwania odpowiednich katalogów w celu ostatecznego ustalenia jego lokalizacji. W celu uniknięcia czasochłonnego przeszukiwania katalogów podczas lokalizowania pliku przy każdej operacji na nim, wprowadzona została funkcja systemowa `open`, której zadaniem jest przydział niezbędnych zasobów w jądrze, umożliwiających wykonywanie dalszych operacji na pliku bez potrzeby przeszukiwania katalogów. Funkcja `open` zwraca *deskryptor*, który jest przekazywany jako parametr aktualny, identyfikujący plik, do funkcji systemowych związanych z operacjami na otwartych plikach. Przy otwieraniu pliku przekazywany jest tryb otwarcia, określający dopuszczalne operacje, jakie można wykonać w ramach tego otwarcia (czyli na uzyskanym deskryptorze), np. *tylko zapis*, *tylko odczyt* lub *zapis i odczyt*. Tryb otwarcia może mieć również wpływ na sposób wykonania tych operacji, np. każda operacja zapisu dopisuje dane na końcu pliku.

`int open(const char *`*pathname*`, int` *flags*`)` — otwarcie pliku. Funkcja zwraca deskryptor otwartego pliku lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
flags	tryb otwarcia:
	`O_WRONLY` — tylko do zapisu,
	`O_RDONLY` — tylko do odczytu,
	`O_RDWR` — do zapisu lub do odczytu.

2.1.2 Zapis i odczyt pliku

Zwrócony przez funkcję `open` (lub inną funkcję systemową) deskryptor służy do identyfikacji pliku w celu wykonania podstawowych operacji dostępu, czyli zapisu lub odczytu. Odczyt pliku polega na skopiowaniu pewnego fragmentu zawartości pliku, określonego z dokładnością do bajtów, do wskazanego obszaru pamięci w przestrzeni adresowej procesu. Odczyt pliku w systemie UNIX realizowany jest przez funkcję systemową `read`. Zapis pliku polega z kolei na umieszczeniu w pliku danych pochodzących z obszaru pamięci w przestrzeni adresowej procesu. Podobnie jak w przypadku odczytu, odpowiedni blok danych określony jest z dokładnością do bajta.

`ssize_t read(int` *fd*`, void *`*buf*`, size_t` *count*`)` — odczyt z pliku. Funkcja zwraca liczbę odczytanych bajtów, lub –1 w przypadku błędu. Zwrócenie wartości 0 oznacza osiągnięcie końca pliku.

Opis parametrów:

fd deskryptor pliku, z którego następuje odczyt danych,

buf adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane,

count liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane).

`ssize_t write(int` *fd*`, const void *`*buf*`, size_t` *count*`)` — zapis do pliku. Funkcja zwraca liczbę zapisanych bajtów, lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor pliku, do którego zapisywane są dane,

buf adres początku obszaru pamięci, zawierającego blok danych do zapisania w pliku,

count liczba bajtów do zapisania w pliku (rozmiar zapisywanego bloku).

Analizując interfejs funkcji `read` lub `write` łatwo zauważyć, że nie ma w nich możliwości wskazania miejsca w pliku, z którego mają pochodzić odczytywane dane lub w którym należałoby te dane umieścić. Miejsce to wyznacza *wskaźnik bieżącej pozycji*, związany z każdym otwartym plikiem. Początkowa wartość tego wskaźnika (zaraz po otwarciu pliku) wynosi 0, co oznacza pierwszy bajt pliku[1]. Operacja odczytu odczytuje zatem określoną liczbę bajtów począwszy do tego wskaźnika. Wskaźnik z kolei przesuwany jest o liczbę odczytanych bajtów w przód (w kierunku końca pliku). W ten sposób kolejna operacja odczytu spowoduje pobranie kolejnego fragmentu zawartości pliku. W przypadku zapisu ten sam wskaźnik wyznacza miejsce, od którego umieszczone zostaną bajt po bajcie zapisywane dane. Jeśli wskaźnik bieżącej pozycji wskazuje miejsce w środku pliku, nowo zapisywane wartości bajtów nadpiszą wartości istniejące. Jeśli wskaźnik bieżącej pozycji wskazuje na koniec pliku lub zapisywany blok nie mieści się w dotychczasowym rozmiarze pliku, nastąpi rozszerzenie pliku, czyli powiększenie jego rozmiaru.

Wskaźnik bieżącej pozycji można zmieniać niezależnie od wykonywania operacji zapisu lub odczytu. Służy do tego funkcja systemowa `lseek`.

`off_t lseek(int` *fd*`, off_t` *offset*`, int` *whence*`)` — przesunięcie wskaźnika bieżącej pozycji. Funkcja zwraca wartość wskaźnika bieżącej pozycji po przesunięciu, liczoną względem początku pliku lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor otwartego pliku,

offset wielkość przesunięcia (wartość ujemna oznacza cofanie wskaźnika, tj. przesuwanie w kierunku początku, a wartość dodatnia oznacza przesuwanie do przodu, tj. w kierunku końca pliku),

whence odniesienie dla przesunięcia, może przyjąć jedną z wartości:
`SEEK_SET` — przesunięcie liczone jest względem początku pliku,
`SEEK_END` — przesunięcie liczone jest względem końca pliku,
`SEEK_CUR` — przesunięcie liczone jest względem bieżącej pozycji.

System UNIX nie udostępnia bezpośredniego mechanizmu rozszerzania pliku przez wstawienie danych wewnątrz pliku z jednoczesnym przesunięciem dotychczasowych danych w kierunku końca. Taka operacja musi zostać zrealizowana w ramach programu użytkownika. Podobnie nie jest dostępny bezpośredni mechanizm usuwania fragmentu znajdującego się wewnątrz pliku. Można natomiast usunąć fragment pliku począwszy od określonego miejsca aż do końca pliku, a dokładniej — zredukować rozmiar pliku do określonej długości. Jest to operacja skracania pliku i realizowana jest przez funkcję systemową `truncate` lub `ftruncate`. Funkcje te umożliwiają skrócenie pliku do długości określonej w bajtach i przekazanej jako drugi parametr aktualny.

[1] Indeksowanie bajtów w pliku rozpoczyna się od wartości 0, podobnie jak tablic w języku C.

`int truncate(const char *`*pathname*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez nazwę. Funkcja zwraca 0 w przypadku pomyślnego zakończenia lub –1 w przypadku błędu.

`int ftruncate(int` *fd*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez deskryptor. Funkcja zwraca 0 w przypadku pomyślnego zakńczenia lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
fd	deskryptor pliku,
length	rozmiar w bajtach, do którego nastąpi skrócenie.

2.1.3 Zamykanie pliku

Zasoby jądra przydzielone na potrzeby dostępu do pliku można zwolnić poprzez wywołanie funkcji systemowej `close`. Funkcja `close` zamyka otwarty plik, a dokładniej deskryptor tego pliku. W przypadku pliku zwykłego deskryptor należy zamknąć wówczas, gdy nie jest używany do wykonywania operacji na pliku w dalszej części programu lub gdy brakuje zasobów jądra (np. wolnych deskryptorów) na potrzeby otwarcia innych plików. **Wszystkie deskryptory otwartych plików, istniejące w danym procesie są zamykane przez system operacyjny wraz z zakończeniem tego procesu**.

`int close(int` *fd*`)` — zamknięcie deskryptora pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd	deskryptor pliku, który zostanie zamknięty.

2.1.4 Tworzenie plików zwykłych

Omówione dotychczas funkcje systemowe dotyczyły plików istniejących już w systemie. Jądro systemu operacyjnego dostarcza też mechanizm tworzenia nowych plików . Mechanizm tworzenia plików zwykłych dostępny jest przez funkcję systemową `creat`, która tworzy plik o nazwie podanej jako parametr aktualny i otwiera utworzony plik w trybie do zapisu, zwracając odpowiedni deskryptor. Jeśli plik o takiej nazwie już istnieje, a proces wywołujący funkcję `creat` ma prawo do zapisu tego pliku, to jego zawartość jest usuwana.

`int creat(const char *`*pathname*`, mode_t` *mode*`)` — utworzenie nowego pliku lub usunięcie jego zawartości, gdy już istnieje oraz otwarcie go do zapisu. Funkcja zwraca deskryptor pliku do zapisu lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
mode	prawa dostępu do nowo tworzonego pliku.

2.1.5 Usuwanie pliku

Istniejący w systemie plik można usunąć za pomocą funkcji `unlink`. Formalnie, funkcja `unlink` usuwa dowiązanie do pliku, czyli wpis katalogowy wyspecyfikowany przez nazwę, podaną jako parametr aktualny. Jeśli jest to ostatnie dowiązanie do danego pliku (w szczególności zatem jedyne dowiązanie), plik usuwany jest z systemu. Jeśli nie jest to ostatnie dowiązanie, zmniejszany jest tylko licznik dowiązań, a plik jest dalej dostępny przez inne nazwy.

`int unlink(const char *`*pathname*`)` — usunięcie dowiązania do pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa).

2.2 Przykłady zastosowania operacji plikowych

Listing 2 przedstawia program do kopiowania pliku. W programie wykorzystano funkcje systemowe `open`, `creat`, `read`, `write` i `close`. Nazwy plików przykazywane są jako argumenty w linii poleceń przy uruchamianiu programu. Jako pierwszy argument przekazywana jest nazwa istniejącego pliku źródłowego, a jako drugi argument przekazywana jest nazwa pliku docelowego, który może zostać dopiero utworzony.

Listing 2: Kopiowanie pliku

```
#include <fcntl.h>
#include <stdio.h>
#define MAX 512

int main(int argc, char* argv[]){
   char buf[MAX];
   int desc_zrod, desc_cel;
   int lbajt;

   if (argc<3){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <plik zrodlowy> <plik docelowy>\n", argv
         [0]);
      exit(1);
   }

   desc_zrod = open(argv[1], O_RDONLY);
   if (desc_zrod == -1){
      perror("Blad otwarcia pliku zrodlowego");
      exit(1);
   }

   desc_cel = creat(argv[2], 0640);
   if (desc_cel == -1){
      perror("Blad utworzenia pliku docelowego");
      exit(1);
   }

   while((lbajt = read(desc_zrod, buf, MAX)) > 0){
      if (write(desc_cel, buf, lbajt) == -1){
         perror("Blad zapisu pliku docelowego");
         exit(1);
      }
   }
   if (lbajt == -1){
      perror("Blad odczytu pliku zrodlowego");
      exit(1);
   }

   if (close(desc_zrod) == -1 || close(desc_cel) == -1){
      perror("Blad zamkniecia pliku");
      exit(1);
   }

   exit(0);
}
```

Opis programu: W liniach 10–14 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik źródłowy i sprawdzana jest poprawność wykonania tej operacji (linie 16–20). Podobnie tworzony jest

i otwierany w trybie *tylko do zapisu* plik docelowy (linie 22–26). Właściwe kopiowanie zawartości pliku źródłowego do pliku docelowego następuje w pętli w liniach 28–33. Wyjście z pętli `while` następuje w wyniku zwrócenia przez funkcję `read` wartości 0 lub –1. Wartość –1 oznacza błąd, co sprawdzane jest zaraz po zakończeniu pętli w liniach 34–37. Po każdym błędzie funkcji systemowej wyświetlany jest odpowiedni komunikat i następuje zakończenie procesu przez wywołanie funkcji systemowej `exit`. Jeśli wywołania funkcji systemowych zakończą się bezbłędnie, sterowanie dochodzi do linii 39, gdzie następuje zamknięcie plików.

Listing 3 przedstawia program do wyświetlania rozmiaru pliku. W programie wykorzystano funkcje systemowe `open`, `lseek` i `close`. Nazwa pliku przykazywana jest jako argument w linii poleceń przy uruchamianiu programu.

Listing 3: Wyprowadzanie rozmiaru pliku

```
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char* argv[]){
   int desc;
   long rozm;

   if (argc < 2){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <nazwa pliku>\n", argv[0]);
      exit(1);
   }

   desc = open(argv[1], O_RDONLY);
   if (desc == -1){
      perror("Blad otwarcia pliku");
      exit(1);
   }

   rozm = lseek(desc, 0, SEEK_END);
   if (rozm == -1){
      perror("Blad w pozycjonowaniu");
      exit(1);
   }

   printf("Rozmiar pliku %s: %ld\n", argv[1], rozm);

   if (close(desc) == -1){
      perror("Blad zamkniecia pliku");
      exit(1);
   }

   exit(0);
}
```

Opis programu: W liniach 8–12 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik o nazwie podanej jako argument w linii poleceń i sprawdzana jest poprawność wykonania tej operacji (linie 14–18). Po otwarciu pliku następuje przesunięcie wskaźnika bieżącej pozycji za pomocą funkcji `lseek` na koniec pliku i zarazem odczyt położenia tego wskaźnika względem początku pliku (linia 20). Uzyskany wynik działania funkcji `lseek`, jeżeli nie jest to wartość –1, jest rozmiarem pliku w bajtach. Wartość ta jest wyświetlana na standardowym wyjściu (linia 26), po czym plik jest zamykany (linia 28).

Listing 4 zawiera rozbudowaną wersję programu z listingu 3, w ten sposób, że wyświetlane są rozmiary wszystkich plików, których nazwy zostały przekazane jako argumenty w linii poleceń.

Listing 4: Wyprowadzanie rozmiaru wielu plików

```
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char* argv[]){
   int desc, i;
   long rozm;

   if (argc < 2){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <nazwa pliku> ...\n", argv[0]);
      exit(1);
   }

   for (i=1; i<argc; i++) {
      desc = open(argv[i], O_RDONLY);
      if (desc == -1){
         char s[50];
         sprintf(s, "Blad otwarcia pliku %s", argv[i]);
         perror(s);
         continue;
      }

      rozm = lseek(desc, 0, SEEK_END);
      if (rozm == -1){
         perror ("Blad w pozycjonowaniu");
         exit(1);
      }

      printf("Rozmiar pliku %s: %ld\n", argv[i], rozm);

      if (close(desc) == -1){
         perror("Blad zamkniecia pliku");
         exit(1);
      }
   }

   exit(0);
}
```

2.3 Deskryptory otwartych plików

Deskryptor otwartego pliku jest indeksem pozycji w *tablicy otwartych plików procesu*, zwanej również *tablicą deskryptorów*. Deskryptory, podobnie jak indeksy tablic w języku C, przyjmują wartości od 0 do wartości o 1 mniejszej od rozmiaru tablicy. Rozmiar tablicy deskryptorów jest najczęściej potęgą liczby 2, np. 64, 128. Rozmiar tablicy deskryptorów limituje liczbę jednocześnie otwartych plików w danym procesie.

Wszystkie funkcje przydzielające deskryptory (np. `open`, `creat`) alokują deskryptor o najniższym wolnym numerze. Programista nie ma bezpośredniego wpływu na przydzielony numer deskryptora i w większości przypadków numer ten nie ma istotnego znaczenia. We wszystkich programach standardowych zakłada się jednak, że pewne deskryptory odgrywają szczególną rolę, polegającą na identyfikacji standardowych strumieni danych w następujący sposób:

- deskryptor nr 0 — standardowe wejście,

- deskryptor nr 1 — standardowe wyjście,
- deskryptor nr 2 — standardowe wyjście awaryjne.

Odczyt danych ze standardowego wejścia sprowadza się do wywołania funkcji `read` na pliku identyfikowanym przez deskryptor nr 0. Podobnie, zapis na standardowe wyjście oznacza wywołanie funkcji `write` na pliku identyfikowanym przez deskryptor nr 1. Deskryptor nr 2 identyfikuje plik, w którym umieszczane są np. komunikaty o błędach.

Szczególna rola wymienionych deskryptorów wynika z faktu, że programy systemowe UNIX'a oraz pewne funkcje biblioteczne przekazują wyniki swojego działania właśnie na standardowe wyjście, czyli do jakiegoś pliku, którego deskryptor ma ustalony numer — 1. Podobnie komunikaty o błędach przekazywane są na standardowe wyjście awaryjne. Tak działają na przykład programy `ls`, `ps`, funkcje biblioteczne `printf`, `perror` itp.

Wykonując zatem w systemie UNIX jakieś polecenie standardowe (np. `ls`, `ps`), uruchamiany jest proces, który zapisuje strumień danych pod deskryptor nr 1. Jeśli standardowe wyjście nie zostanie przekierowane, dane trafią na terminal. Terminal jest plikiem specjalnym, stanowiącym domyślne standardowe wyjście większości poleceń systemowych. Jeśli uruchomimy polecenie z przekierowaniem standardowego wyjścia do pliku (np. `ls > lista.txt`), w odpowiednim procesie przed rozpoczęciem wykonywania programu danego polecenia wskazany plik zostanie otwarty do zapisu i umieszczony pod deskryptorem nr 1.

Programy `ls` i `ps` nie pobierają żadnych danych wejściowych (jedynie argumenty i opcje przekazane w linii poleceń), istnieje natomiast duża grupa programów, które na standardowe wyjście przekazują wyniki przetwarzania danych wejściowych. Typowymi przykładami takich programów są *filtry*: `more`, `grep`, `sort`, `tr`, `cut` itp. Plik z danymi wejściowymi dla tych programów może być przekazany przez podanie jego nazwy jako jednego z argumentów w linii poleceń. Jeśli jednak filtr zostanie użyty bez nazwy pliku w argumentach linii poleceń, proces będzie próbował czytać dane ze standardowego wejścia, czyli otwartego pliku o ustalonym numerze deskryptora — 1. Domyślnym standardowym wejściem jest również terminal, chociaż można na nie skierować np. plik zwykły (`cat < lista.txt`).

Warto w tym miejscu podkreślić, że z punktu widzenie programu nie jest istotne, jaki plik lub jaki rodzaj pliku identyfikowany jest przez dany deskryptor. Ważne jest, jakie operacje można na takim pliku wykonać. W ten sposób przejawia się niezależności plików od urządzeń. Najczęściej wykonywanymi funkcjami na plikach identyfikowanych przez deskryptory 0–2 są `read` i `write`. Warto też zwrócić uwagę na fakt, że funkcja systemowa `lseek` może być wykonywana na pliku o dostępie bezpośrednim (swobodnym), nie może być natomiast wykonana na pliku o dostępie sekwencyjnym, czyli urządzeniu lub łączu komunikacyjnym. Za pomocą `more` można więc cofać się w przeglądanym pliku tylko wówczas, gdy jego nazwa jest przekazana jako parametr w linii poleceń, gdyż tylko wówczas jest on otwierany przez funkcję `open` w procesie `more` i traktowany jest jako plik o dostępie bezpośrednim. W przypadku odczytu danych ze standardowego wejścia plik otwierany jest w procesie powłoki i przekazywany do procesu `more`, który nie wie, z jakiego typu strumieniem ma do czynienia. Wyodrębnienie funkcji `lseek` i ograniczenie jej stosowalności wyłącznie do plików o dostępie bezpośrednim umożliwia zachowanie takiego samego interfejsu dostępu do zawartości pliku (funkcje `read` i `write`) zarówno w przypadku plików o dostępie sekwencyjnym, jak i bezpośrednim.

W związku ze szczególnym charakterem deskryptorów standardowych strumieni danych może okazać się konieczne wymuszenie przydziału takiego deskryptora do jakiegoś pliku tak, żeby plik ten pełnił określoną rolę dla danego procesu, np. rolę standardowego wejściu lub standardowego-wyjścia. Jak już wcześniej wspomniano, nie można wymusić przydziału deskryptora bezpośrednio, można natomiast zmienić numer deskryptora danego pliku już po jego przydzieleniu. W celu zmiany deskryptora należy go najpierw powielić, czyli uzyskać duplikat deskryptora pod oczekiwanym numerem, a następnie za pomocą funkcji systemowej `close` zamknąć dotychczasowy deskryptor, jeśli nie jest już potrzebny. Powielenie deskryptorów umożliwiają funkcje systemowe `dup` i `dup2`.

Funkcja `dup` przydziela deskryptor zgodnie z zasadą alokacji najniższego wolnego numeru. Jeśli ma to być oczekiwany numer, należy zagwarantować, że w momencie wywołania funkcji

`dup` jest on najniższym wolnym numerem. Przykład takiego programu znajduje się na listingu 8 (str. 22)

Większą funkcjonalność oferuje `dup2`, w przypadku której można wskazać numer duplikatu, specyfikując go jako drugi parametr. Jeśli wyspecyfikowany numer jest zajęty nastąpi zamknięcie odpowiadającego mu deskryptora i dopiero wówczas utworzenie duplikatu.

`int dup(int` *fd*`)` — powielenie (duplikacja) deskryptora. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.
Opis parametrów:
fd deskryptor, który ma zostać powielony.

`int dup2(int` *oldfd*`, int` *newfd*`)` — powielenie (duplikacja) deskryptora we wskazanym miejscu w tablicy deskryptorów. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.
Opis parametrów:
oldfd deskryptor, który ma zostać powielony,
newfd numer nowo przydzielanego deskryptora.

Zastosowanie funkcji `dup2` do przekierowania standardowych strumieni danych pokazane zostało wraz z użyciem łącza nienazwanego na listingu 3 (str. 29) i 4 (str. 30).

2.4 Zadania

2.1. Napisać program do rozpoznawania, czy plik o nazwie podanej jako argument linii poleceń jest plikiem tekstowym.
Wskazówka: wykorzystaj fakt, że plik tekstowy zawiera znaki o kodach 0–127 (można w tym celu użyć funkcji `isascii`).

2.2. Napisać program konwertujący małe litery na duże w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia, a wyniki zapisane na standardowe wyjście.
Wskazówka: odczytaj blok z pliku do bufora, sprawdź kody poszczególnych znaków i jeśli odpowiadają małym literom, to dodaj do kodu znaku różnicę pomiędzy kodem litery `A` i `a` (można też użyć funkcji `toupper`), a następnie zapisz zmodyfikowany blok w tym samym miejscu pliku, z którego był odczytany (cofnij odpowiednio wskaźnik bieżącej pozycji) lub na standardowym wyjściu, jeśli dane pochodziły ze standardowego wejścia.

2.3. Napisać program ustalający liczbę znaków w najdłuższej linii w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.4. Napisać program wyświetlający najdłuższą linię w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.5. Napisać program, który w pliku o nazwie podanej jako ostatni argument zapisze połączoną zawartość wszystkich plików, których nazwy zostały podane linii poleceń przed ostatnim argumentem.

2.6. Napisać program do wyznaczania częstości występowania liter w pliku tekstowym o nazwie podanej jako argument linii poleceń. Wynikiem działania programu powinien być wydruk na standardowym wyjściu określający procentową zawartość poszczególnych liter w całym tekście z pominięciem białych znaków oraz znaków interpunkcji. Jeśli nazwa pliku nie została podana, tekst powinien zostać odczytany ze standardowego wejścia.

2.7. Napisać program, który policzy wszystkie słowa w pliku o nazwie podanej jako argument linii poleceń, przyjmując, że słowa składają się z małych i dużych liter alfabetu oraz cyfr i znaku podkreślenia, a wszystkie pozostałe znaki są separatorami słów. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.8. Napisać program, który wyodrębni wszystkie słowa w pliku o nazwie podanej jako argument linii poleceń oraz zapisze wyodrębnione słowa na standardowym wyjściu, oddzielając je znakiem nowej linii. Założyć, że słowa składają się z małych i dużych liter alfabetu oraz cyfr i znaku podkreślenia, a wszystkie pozostałe znaki są separatorami słów. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.9. Napisać program do porównywania dwóch plików o nazwach przekazanych jako argumenty linii poleceń. Wynikiem działania programu ma być komunikat, że *pliki są identyczne*, *pliki różnią się począwszy od znaku nr <numer znaku> w linii <numer linii>* lub — gdy jeden z plików zawiera treść drugiego uzupełnioną na końcu o jakieś dodatkowe znaki — *plik <nazwa> zawiera <liczba> znaków więcej niż zawartość pliku <nazwa>*.

2.10. Napisać program do sortowania w porządku alfabetycznym linii pliku, którego nazwa została przekazana jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia a wynik zapisany na standardowym wyjściu.

2.11. Napisać program do filtrowania linii tekstu odczytywanego ze standardowego wejścia w taki sposób, że jeśli linia odczytanego tekstu zawiera łańcuch znaków przekazanych jako argument linii poleceń, to jest ona zapisywana na standardowym wyjściu, w przeciwnym przypadku jest pomijana.

2.12. Napisać program do znajdowania łańcucha znaków podanego jako pierwszy argument linii poleceń w plikach o nazwach podanych jako pozostałe argumenty linii poleceń. Program powinien informować o nazwie pliku i miejscu (numer linii, numer znaku w linii), w którym dany łańcuch został znaleziony.

2.13.* Napisać program do formatowania akapitów tekstu w plikach o nazwach podanych jako argumenty linii poleceń w taki sposób, żeby długość linii (liczba znaków w linii) nie przekraczała wartości podanej jako pierwszy argument linii poleceń. Przyjąć, że w wyniku formatowania nie może nastąpić podział słowa pomiędzy dwa wiersze i że separatorem formatowanego akapitu jest pusta linia.

2.14.* Zaimplementować mechanizm przyspieszający dostępu do danych w pliku `/etc/passwd` za pomocą pliku indeksowego o strukturze B-drzewa lub B+-drzewa.

2.15.* Zaimplementować mechanizm przyspieszający dostępu do danych w pliku `/etc/passwd` za pomocą tablicy hashowej.

2.5 Pytania kontrolne

1. Jaką wartość zwraca funkcja systemowa w przypadku błędu w jej wykonaniu?
2. Jakie możliwości dostępu do plików oferuje jądro systemu operacyjnego UNIX?
3. Jakie parametry należy przekazać do funkcji `open` w celu otwarcia pliku?
4. Co jest wartością zwrotną funkcji `open`?
5. W jakim trybie można otworzyć plik?
6. Co służy do identyfikacji pliku w celu wykonania operacji odczytu (read) lub zapisu (write)?
7. Jaka jest zasada przydziału numeru deskryptora pliku?

8. Jakie numery deskryptorów przypisane są do standardowych strumieni danych?
9. Jakie funkcje systemowe służą do duplikowania deskryptorów plików?
10. Jakiej funkcji systemowej należy użyć w celu zwolnienia miejsca w tablicy deskryptorów?
11. Kiedy następuje zamknięcie deskryptora otwartego pliku?
12. Jaka funkcja systemowa służy do tworzenia pliku?
13. Co robi funkcja systemowa `creat`?
14. W jakim trybie funkcja systemowa `creat` otwiera plik?
15. Co się dzieje z istniejącym plikiem przy próbie jego utworzenia za pomocą funkcji `creat`?
16. Od jakiego miejsca w pliku odczytywany jest blok danych przez funkcję systemową `read`?
17. Od jakiego miejsca w pliku zapisywany jest blok danych przez funkcję systemową `write`?
18. Jakiej wielkości blok odczytywany jest przez funkcję systemową `read`?
19. Jakiej wielkości blok zapisywany jest przez funkcję systemową `write`?
20. Jaka funkcja systemowa służy do zmiany wskaźnika bieżącej pozycji w pliku?
21. Jaka jest wartość wskaźnika bieżącej pozycji zaraz po otwarciu pliku?
22. Względem czego można określić ustawienie wskaźnika bieżącej pozycji w pliku za pomocą funkcji `lseek`?
23. Jakie parametry należy przekazać do funkcji `lseek`?
24. W jaki sposób można uzyskać rozmiar otwartego pliku?