Notatki Systemy Operacyjne 1

Piotr Kotynia

10 listopada 2021

Notatki studenckie zrealizowane na podstawie wykładów mgr inż. Pawła Sobótki uzupełnione o wiedzę ze slajdów, manuala, internetu i własną interpretację obejmujące raczej teorytyczną wiedzę na przedmiot Systemy Operacyjne 1. W wielu przypadkach nie opisywałem obszernie działania, parametrów i wartości zwracanych funkcji zostawiając to do doczytania w manualu. Notatki mogą być niekompletne(choć się starałem), potencjalnie zawierać błędy i niepoprawne uproszczenia.

Spis treści

1 Systemy operacyjne i komputerowe - wstęp

1.1 System operacyjny, a system komputerowy

1.1.1 Co to jest system operacyjny?

- Program pośredniczący między użytkownikiem i komputerem
- Dystrybutor zasobów - przydziela zasoby systemu i zarządza nimi
- Program sterujący - kontroluje wykonanie programów użytkownika oraz pracę urządzeń wejścia/wyjścia.
- **Jądro**(kernel) - jedyny program działający przez cały czas
- Podstawowe oprogramowanie systemu komputerowego, które pozwala
  1. wykonywać programy użytkownika i ułatwiać rozwiązywanie powstających problemów
  2. uczynić system komputerowy wygodnym w używaniu
  3. wykorzystać sprzęt jak najbardziej efektywnie
  4. Zarządzać sprzętowymi i programowymi zasobami systemu komputerowego
  5. Przekształcać maszyny rzeczywistej w maszynę wirtualną o cechach wymaganych przez przyjęty tryb przetwarzania

1.1.2 Składowe systemu komputerowego

Sprzęt (hardware) – dostarcza podstawowych zasobów systemowi (procesor, pamięć, urządzenia wejścia/wyjścia).

System operacyjny – zarządza i koordynuje wykorzystanie sprzętu przez różnorodne programy aplikacyjne użytkowników.

Programy aplikacyjne – określają w jaki sposób należy użyć zasobów systemu dla rozwiązania zadań określonych przez użytkownika (kompilatory, systemy baz danych, gry, programy biurowe).

Użytkownicy – ludzie, maszyny, inne komputery.

1.1.3 Tryby pracy systemu komputerowego

Pośredni – wsadowy (offline, batch) zadania od poszczególnych użytkowników gromadzone są na nośniku jako wsad i wykonywane jedno po drugim w określonej kolejności. Brak wielozadaniowości. Dalej używane np. dla dużych obliczeń.

Bezpośredni – interakcyjny (on-line) symuluje wykonywanie wielu zadań jednocześnie (każdy PC, system z short-time schedulerem). Procesor jest przełączany pomiędzy kilkoma zadaniami, które są przechowywane w pamięci operacyjnej i na dysku.

W czasie rzeczywistym [1] – (real-time) Mają dobrze określone, stałe ograniczenia czasowe odpowiedzi na zewnętrzny bodziec.

System rygorystyczny (Hard RT system) Posiadają gwarantowane, nieprzekraczalne ograniczenia czasowe. Sztywne gwarancje czasowe eliminują konstrukcje zapewniające zmienność czasu realizacji, redukując w ten sposób koszty. Stosowane przede między innymi w systemach bezpieczeństwa (lotnictwo, szlabany, linie produkcyjne).

System łagodny (Soft RT system) Wersja łagodna, posiada limity czasu, ale pod pewnymi warunkami mogą być przekroczone. Wykorzystywane głównie w multimediach (np. VR).

[1] W tym kursie będzie jedynie zmianka o systemach RT, jest to bardzo rozbudowana dziedzina

1.2 Zadania systemów operacyjnych

Efektywne zarządzanie zasobami systemu komputerowego

- Przydział i odzyskiwanie zasobów
- Planowanie dostępu do zasobów
- Ochrona i autoryzacja dostępu do zasobów
- Rozliczanie użytkowników z wykorzystania zasobów
- Obsługa błędów

Zasoby systemu komputerowego

- Procesor(y), rdzenie
- Pamięć i inne urządzenia systemu komputerowego
- Informacja przchowywana w systemie

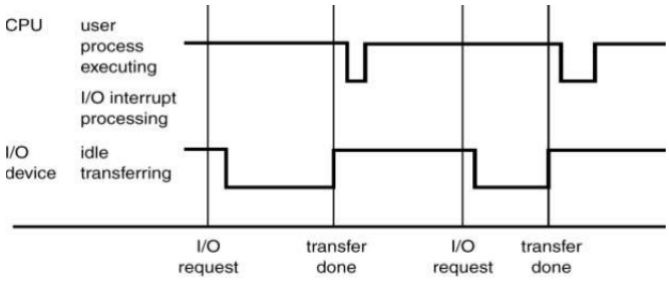
1.3 Działanie systemu komputerowego

Urządzenia wejścia/wyjścia i procesor mogą pracować współbieżnie, współzawodnicząc w dostępie do pamięci. **Sterownik urządzenia** (device controller) zarządza urządzeniami określonego typu i nadzoruje operacje wejścia/wyjścia pomiędzy urządzeniem, a lokalnym buforem sterownika urządzenia. Sterownik urządzenia powiadamia procesor o zakończeniu operacji wejścia/wyjścia za pomocą przerwania (interrupt).

1.3.1 Przerwania

Co to jest przerwanie? Przerwanie to zdarzenie, które powoduje, że potok przetwarzania procesora (wykonywanie instrukcji) jest przerywany i sterowanie przekazane jest do procedury obsługi przerwań (interrupt handler), czyli funkcji zaimplementowanej w kernelu). Adresy różnych interrupt handlerów znajdują się w wektorze przerwań (interrupt vector). Gdy jakiś proces jest przerywany, architektura zwykle blokuje przychodzenie nowych przerwań, choć są wyjątki.

Pułapka (trap) – innaczej wyjątek. Jest rodzajem przerwania generowanym programowo dla sygnalizacji błędu (np. dzielenia przez zero) bądź żądania realizacji zamówienia, wymagającego obsłużenia przez system operacyjny.



Rysunek 1: Uproszczony model obsługi przerwania przez CPU

Obsługa przerwań System operacyjny zachowuje stan procesora (rejestry, licznik rozkazów). W wirtualnym pliku /proc/interrupts możemy odczytać statystyki przerwań w systemie

1.3.2 Obsługa wejścia/wyjścia

Są dwa rodzaje obsługi:

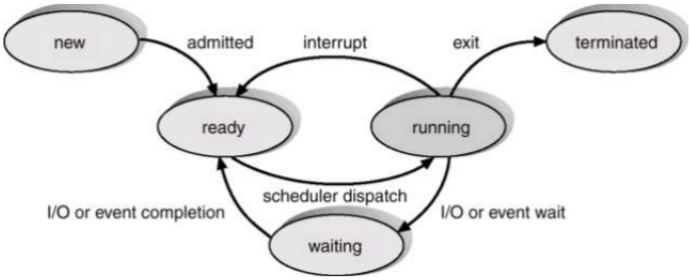
Synchroniczna operacja wejścia/wyjścia Proces, który chce wykonać operacje wejścia wyjścia będzie czekał po wysłaniu prośby o wywołanie systemowe aż dostanie odpowiedź zwrotną. W tym czasie nie będzie wykonywał żadnych operacji.

Asynchroniczna operacja wejścia/wyjścia Proces, który wykonuje operacje wejścia wyjścia natychmiast po wysłaniu prośby, dostaje sterowanie z powrotem. W trakcie wykonywania operacji jest w stanie wykonywać dalej różne czynności. Przykład: API drivera karty graficznej otrzymuje request wyrenderowania klatki, a w tym czasie procesor jest wolny i może przygotować informacje do wyrenderowania następnej klatki.

2 Procesy

2.1 Koncepcja procesu

Proces – (zadanie) wykonujący się program. Procesy są rozróżniane za pomocą identyfikatorów procesów **(PID - process identifier)**, które są liczbami całkowitymi. Jeden program może tworzyć wiele procesów.



Rysunek 2: Diagram stanu procesów. Scheduler przypisuje procesowi stan running, proces może sam się zakończyć (exit), być wywłaszczony przez przerwanie (interrupt) lub samemu wejść w stan oczekiwania na operacje wejścia wyjścia (waiting)

2.1.1 Składowe procesu

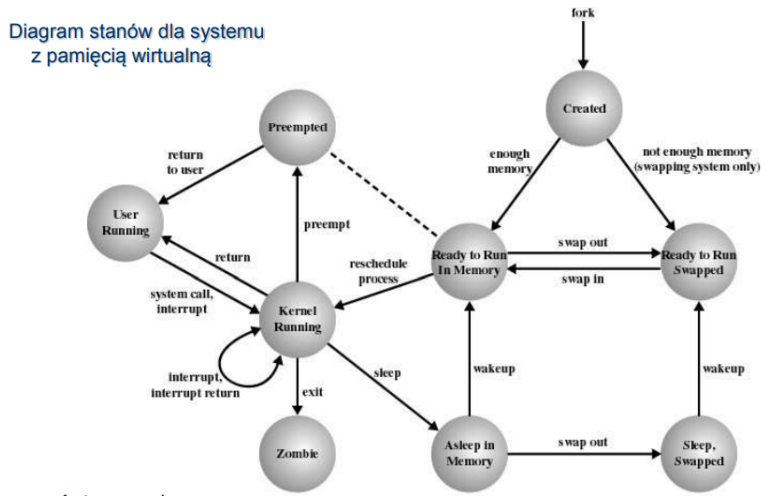
- Kod (text section) – licznik rozkazów i rejestry procesora
- Stos (stack) – zawiera tymczasowe dane: parametry wywołania funkcji, zmienne lokalne (automatyczne)
- Sekcja danych – zawiera zmienne globalne i statyczne
- Sterta (heap) – zawiera dane przydzielane dynamicznie

2.1.2 Stan procesu

- Nowy (*new*) – proces został utworzony
- Aktywny (running) – są wykonywane instrukcje procesu
- Oczekujący (waiting) – proces czeka na wystąpienie jakiegoś zdarzenia
- Gotowy (ready) – proces czeka na przydział procesora
- Zakończony (terminated) – proces zakończył działanie

2.1.3 Blok kontrolny procesu (PCB)

Jest to tak naprawdę struktura w C, która zawiera informacje na temat procesu, implementacja znajduje się w */include/linux/sched.h* (polecam przeczytać.) Informacje zawarte w bloku kontrolnym procesu (process control block - PCB):



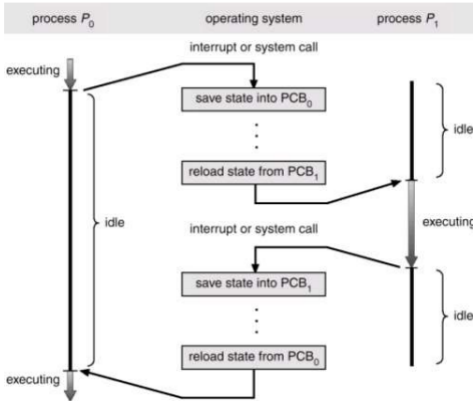
Rysunek 3: Rozbudowany diagram z pamięcią wirtualną. Istotna cecha – swap in / swap out - operacja zapisania danych z procesu do pamięci trwałej, używana zwykle przy niedoborach pamięci operacyjnej

- stan procesu
- licznik rozkazów
- rejestry procesora
- informacje o planowaniu przydziału procesora
- informacje o zarządzaniu pamięcią
- informacje do rozliczeń
- informacje o stanie wejścia/wyjścia

2.1.4 Przełączanie procesora między procesami

Kernel zapisuje Przyczyny przerwania wykonania procesu:

- przerwanie zegarowe
- przerwania od urządzeń
- wywołanie f. systemowej
- wystąpienie pułapki



Rysunek 4: Schemat zamiany obsługiwanego procesu nazywany **zmianą kontekstu**. Zajmuje się tym scheduler (pol. planista, dyspozytor). W momencie kiedy scheduler ładuje stan procesu do rejestrów procesora to również konfiguruje hardwareowy timer, który po pewnym czasie (w windowsie 16ms) wywoła przerwanie, wtedy scheduler będzie mógł podjąć decyzje czy ponownie zmienić kontekst

2.2 Planowanie procesów

2.2.1 Kolejki planowania procesów

Procesy w postaci struktur PCB są umieszczane w kolejkach, które obsługuje scheduler. Przykłady kolejek:

- Kolejka zadań (job queue) – zawiera wszystkie zadania w systemie.
- Kolejka zadań gotowych (ready queue) – zawiera wszystkie procesy gotowe do działania (w pamięci operacyjnej)
- Kolejki do urządzeń (device queues) – listy procesów oczekujących na obsługę przez konkretne urządzenia

2.3 Działania na procesach

2.3.1 Tworzenie procesu

Procesy macierzyste (parent processes) tworzą procesy **potomne** (children), które też tworzą podprocesy. W rezultacie powstaje **drzewo procesów**.

Nowe procesy są tworzone za pomocją funkcji systemowej *fork()*, kopiuje ona całą przestrzeń adresową procesu, w którym wywołujemy funkcję. Zwraca: 0 dla procesu dziecka, PID procesu dziecka dla rodzica i -1 dla rodzica gdy nie można było utworzyć procesu dziecka. Uwaga: nigdy nie można przewidzieć, który proces zacznie się wykonywać pierwszy po wykonaniu fork().

Przykładowe polecenia/funkcje do zarządzania procesami:

- *ps* – wypisuje wszystkie procesy w systemie
- *pstree* – wypisuje drzewo procesów
- *getpid()* – zwraca PID procesu
- *getppid()* – zwraca PID rodzica
- *wait(int *stat_loc)* – w spobób synchroniczny oczekuje na zakończenie dowolnego procesu dziecka, jeśli się powiedzie zwraca PID dziecka. Opcjonalnie jeśli podamy stat_loc, zapisze tam status wyjściowy procesu dziecka.
- *waitpid(int *stat_loc, int options)* – wait tylko że inny

2.3.2 Planiści (schedulery)

Podział procesów:

- **ograniczone przez wejście wyjście** – spędzające dużo więcej czasu na wykonywanie operacji wejścia/wyjścia niż na obliczenia (wiele krótkich faz procesora)
- **ograniczone przez dostęp do procesora** – sporadycznie generujące zamówienia na operacje wejścia/wyjścia (nieliczne ale długie fazy procesora)

Rodzaje schedulerów (planistów) :

- **krótkoterminowy** – short-term scheduler/CPU scheduler. Wybiera do wykonania jeden z procesów gotowych i przydziela mu procesor. Podejmuje działanie bardzo często (n.p. co 100ms), więc musi działać szybko.
- **długoterminowy** – long-term scheduler/job scheduler. Wybiera procesy do kolejki procesów gotowych, wywoływany dosyć rzadko (co sekundy, minuty), więc może działać powoli. Kontroluje stopień **wieloprogramowości**. Usiłuje realizować dobrą mieszankę procesów. Nie wszystkie systemy operacyjne mają planistę długoterminowego.

2.3.3 Kończenie procesu

Proces za pomocą funkcji *exit()* prosi aby system operacyjny go zakończył. Kod wyjścia jest przekazywany do procesu macierzystego przez funkcje *wait()*. Proces, na którego rodzic nie "poczekał" funkcją *wait()* to **zombie**.

Proces, którego rodzic zakończył działanie (np. exit()), nie czekając aż procesy dzieci się zakończą, to **sierota** (orphant). Procesy sieroty adoptuje główny proces - **init** o PID 1. [2]

Proces macierzysty może spowodować zakończenie innego procesu (zazwyczaj potomka) za pomocą funkcji systemowej *kill()*

2.4 Środowisko wykonania procesu

Częścią środowiska wykonania procesu UNIX są **zmienne środowiskowe** (environment variables) w postaci: nazwa=wartość. Każdy proces ma swój zestaw zmiennych środowiskowych i **zawsze** mają formę **słownika**: lista w postaci klucz - wartość (string,string). W katalogu */proc* system zakłada podkatalogi z wirtualnymi plikami(nie są fizycznie plikami) odpowiadające informacjom dla każdego procesu. Np. w **environ** są zmienne środowiskowe (ale tylko w momencie początkowym uruchomienia procesu).

Często spotykane zmienne środowiskowe:

- **PATH** - lista ścieżek dostępu do plików wykonywalnych realizujących polecenia powłoki
- **HOME** - katalog macierzysty użytkownika
- **PWD** - aktualny katalog
- **PS1,PS2** - pierwszy i drugi tekst zachęty
- **TERM** - nazwa (typ, model) używanego terminala
- **SHELL** - używana powłoka
- **LOGNAME** -nazwa użytkownika
- **RANDOM** - liczba losowa
- **EDITOR** - edytor użytkownika
- **PPID** - nr procesu rodzicielskiego

[2] W niektórych systemach zamiast procesu init występuje proces **systemd**

Zmienne środowiskowe są dziedziczone przez proces potomny przy *fork()* i definiowane na nowo przy *execve() i execlve()*

Do nadawania wartości zmiennym środowiskowym w powłoce bash służy *name=value; export name*[3]

Aby nadawać wartości zmiennym środowiskowym w programie należy zdefiniować, a następnie modyfikować *extern char **environ*. Funkcje: *getenv(), putenv(), unsetenv() itp.* służą do przystępnej modyfikacji zmiennej environ (zalecane przeczytać manuala).

3 Interfejs systemu plików, strumieniowe wejście/wyjście

3.1 Koncepcja pliku

Plik to logiczna jednostka magazynowania informacji. W systemach UNIX jest to ZAWSZE jedynie tablica bajtów - nic ponad to.

Atrybuty plików:

- **Nazwa** – jedyna informacja przechowywana w postaci czytelnej bezpośrednio przez człowieka
- **Typ** – wymagany przez niektóre systemy operacyjne (dla interpretacji zawartości). Typ pliku może być rozpoznawany przez system, użytkownika bądź aplikację
- **Położenia** – wskaźnik do urządzenia i położenia pliku na tym urządzeniu
- **Rozmiar** – bieżący rozmiar pliku
- **Ochrona** – informacje służące do sprawdzania, kto może plik czytać, zapisywać, wykonywać
- **Czas, data, id użytkownika** – dane służące do ochrony, bezpieczeństwa i doglądania użycia plików

3.1.1 Operacje plikowe

- Create – tworzenie pliku

[3]Samo nadanie name=value nie wystarczy, ponieważ zmieni on tylko zmienną w obrębie procesu, aby wprowadzić ją do środowiska i przekazać do procesu potomnego trzeba go eksportować

- write – zapisywanie do pliku
- read – czytanie z pliku
- file seek – zmiana bieżącej pozycji w pliku
- delete – usuwanie pliku (bądź jego dowiązania do pozycji katalogowej; znaczenie bywa różne)
- truncate – skracanie pliku
- fd=open(Fi) – znajduje w strukturze katalogowej dysku wpis pliku Fi i kopiuje zawartość tego wpisu do pamięci – jeśli pozwalają na to reguły dostępu dla danego procesu. Operacja tworzy nową sesję plikową. Deskryptor fd reprezentuje dostęp do pliku w ramach sesji plikowej.
- Close (fd) – przepisuje zawartość struktury opisującej sesję plikową, związaną z deskryptodem fd, z pamięci do struktury katalogowej pliku (Fi) na dysku.

3.1.2 Blokady dostępu do plików

Systemy operacyjne często udostępniają procesom możliwość zakładania czasowej blokady dostępu do części bądź całego pliku

Blokada obowiązkowa – jest wymuszana przez jądro. Założenie takiej blokady powoduje, że system odmawia realizacji dostępu innym procesom przy próbie dostępu. Wbrew pozorom raczej rzadko używana.

Blokada doradzana – nie jest wymuszana przez jądro. Proces może sprawdzić, czy blokada jest założna przez inny proces, ale respektowanie blokady zależy od programisty.

3.1.3 Otwarte pliki

System utrzymuje w pamięci operacyjnej szereg struktur danych służących do obsługi otwartych plików:

- Wskaźnik bieżącej pozycji, indywidualny dla każdej sesji plikowej
- Licznik otwarć pliku – pozwalający na usunięcie wpisu pliku z tablicy otwartych plików, gdy licznik osiąga wartość 0

- Kopia informacji pozwalającej na odszukanie zawartości pliku na urządzeniu fizycznym
- Struktura informująca o prawach dostępu do pliku

Sesja plikowa – ciąg operacji na pliku pomiędzy otwarciem, a zamknięciem dostępu do pliku. Sesja jest skojarzona z deskryptorem pliku, stanowi on identyfikator sesji plikowej.

3.2 Struktura katalogowa plików

3.2.1 Rodzaje struktur katalogowych

Istnieje kilka sposobów na organizowanie struktury katalogów, każdy z nich ma swoje wady i zalety:

- **Katalog jednopoziomowy** – jeden katalog dla wszystkich użytkowników i wszystkich plików
- **Katalog dwupoziomowy** – Oddzielny katalog dla każdego użytkownika, ale wewnątrz katalogu użytkownika nie ma już żadnych katalogów
- **Katalog o strukturze drzewa** – Efektywne ułożenie plików pozwalające na hierarchizację
- **Katalog o strukturze acyklicznego grafu** – Usprawnienie katalogu o strukturze drzewa dodające możliwość dzielenia dostępu do plików i katalogów. Np. w dwóch katalogach może się znajdować plik, który jest tak naprawdę tym samym plikiem.

3.2.2 Katalog o strukturze acyklicznego grafu

Tworzenie dwócuh nazw (ścieżek) do tego samego pliku czy katalogu nazywamy *aliasingiem*. W UNIXie używamy mechanizmu hard/symlinków (polecenie ln) do tworzenia połączeń w systemie plików. Jeśli usuniemy ścieżke dostępu do pliku wraz z tym plikiem mamy problem – ine ścieżki dostępu przestają być ważne (nazywamy to *dangling pointers*.

Jak zagwarantować, że nie ma cykli?

- Zezwalać na dowiązania (link) do plików a nie do katalogów
- Odśmiecać system plików (garbage collection)

- Przy każdym tworzeniu dowiązania uruchamiać algorytm wykrywania cykli

3.2.3 Montowanie podsystemu plików

System plików musi być **montowany** w systemie operacyjnym zanim może być użyty - czyli wybieramy mu miejsce w grafie gdzie go podepniemy. Niezamontowany podsystem plików jest montowany w **punkcie montażu** (mount point). Poprzednia zawartość jest PRZESŁANIANA przez zamontowane poddrzewo plików - oznacza to, że gdy odepniemy nową zawartość, stara wróci na swoje miejsce. Zazwyczaj podsystem plików montuje się do pustego katalogu (taki korzeń). Za odpowiednio skonfigurowane podmontowanie wszystkich systemów plików odpowiedzialny jest kernel.

Polecenie *mount* wypisuje nam wszystkie podmontowane systemy plików.

3.2.4 Ochrona plików

Najczęściej uzależnia się dostęp do plików od identyfikacji użytkownika, bądź jego roli (role-based access controll). Najczęśniej stosuje się wykaz dostępów dla pliku (access list), zawierający id użytkowników i dozwolone rodzaje dostępu. Dla uproszczenia wprowadza się klasy (grupy) użytkowników.

Specjalnymi grupami w linuxie są **cgroups** (control groups). Jest to system przez który jądro ogranicza dostęp do zasobów (CPU, pamięć itp.) dla wybranych grup.

Tryby dostępu: read, write, execute (RWX). Tryb dostępu da się zakodować za pomocą liczby binarnej: R-4, W-2, E-1, czyli dostęp 7 oznacza wszystkie możliwe operacje. Dla pliku istnieją trzy klasy użytkowników i to dla nich ustalamy poszczególne rodzaje dostępu, są nimi: dostęp właściciela, dostęp grupy i dostęp publiczny.

Dostęp do pliku ustalamy za pomocą polecenia *chmod*.

3.3 Input/Output

3.3.1 Strumieniowe wejście/wyjście

Typ **FILE** jest definiowany przez STANDARD JĘZYKA, a nie przez system. Czyli API niskopoziomowe dla plików jest dostępne jeśli system udostępnia kompilator C zgodny ze standardem.

Deskryptor pliku (file descriptor) – unikalna, nieujemna liczba całkowita służąca do identyfikacji otwartego pliku. Maksymalna wartość i tymsamym maksymalna liczba otwartych plików dla procesora opisuje stała **OPEN_MAX**.

Strumień (stream) – Jest to logiczny objekt, który służy do komunikacji z otwartym plikiem. Maksymalną liczbę otwartych strumieni definiuje zmienna **FOPEN_MAX**. W standardzie ISO są one traktowane jak *FILE**, czyli wskaźniki do plików (file pointers). Do otwierania strumieni, służy np. funkcja *fopen()* zwracająca wskaźnik do obiektu kontrolującego strumień (FILE*).

3.3.2 Buforowanie strumieni

Strumienie mogą być:

- **Niebuforowane** (Unbuffered (_IONBF)) – znaki pisane lub czytane z takiego strumienia są przesyłane oddzielnie tak szybko jak to możliwe
- **Liniowo buforowane** (Line buffered (_IOLBF)) – znaki są przesyłane grupowo po odczytaniu znaku nowej linii (np. strumień terminala)
- **W pełni buforowane** (Fully buffered (_IOBF)) – bajty są wysyłane jako blok, kiedy zapełni się bufor strumienia (domyślnie większość nowo otwartych strumieni)

Dobry rozmiar bufora (*BUFSIZ*) jest podany w *¡stdio.h¿* (cokolwiek to znaczy).

Aby zmienić ustawienie ustawienie bufora, używamy funkcji *int setvbuf(FILE *stream, char *buf, int mode, size_t size)*. Np. *setvbuf(stdout, _IONBF, NULL, 0)* ustawi nam standardowy strumień wyjściowy na tryb niebuforowany, przez co następujący fragment kodu:

```
printf("Hello");
sleep(3);
printf("world\\n");
```

będzie wypisywał najpierw "Hello", potem czekał 3 sekundy, a następnie wypisywał "world
n". W przypadku domyślnego bufora, Tekst wypisał by się cały, dopóki bo drugim wywołaniu funkcji *printf*

Po co strumienie są domyślnie buforowane? Aby zaoszczędzić zasoby. Gdyby przy każdym wypisanym znaku system musiałby przechodzić w tryb jądra, generować przerwania, zmieniać kontekst, etc. Zamiast tego, znaki są wysyłane grupowo.

Funkcja *fflush()* służy do wysłania zbufurowanego outputu do pliku i czyści bufor.

W folderze */proc/[desktyptor procesu]/fd* znajdują się pliki (symlinki) odpowiadające otwartym strumieniom plików. Numery 0,1 i 2 to strumienie domyślne: stdin, stdout i stderr w konsoli.

3.3.3 Blokowanie strumieni, EOF i błędy

Strumienie można blokować np. po to by uniemożliwić dostęp do strumienia innym procesom, służą do tego funkcje: *flockfile(), ftrylockfile(), funlockfile()*

Strumienie wejściowe mogą się skończyć (np. plik się skończył), funkcja *feof()* przyjmuje strumień pliku i sprawdza czy wskaźnik końca pliku (end-of-file indicator) jest ustawiony na plik. Jeśli tak - zwraca niezerową wartość.

Funkcja *ferror()* działa jak funkcja *feof()*, ale dla wskaźnika błędu (error indicator). Sytuacja trudna do wygenerowania, dzieję się przy błędzie przy niskopoziomowej operacji np. zapisu na dysk.

3.3.4 Pozycja strumienia

Dla niektórych strumieni dozwolone jest "skakanie" po pozycjach, do tego służy funkcja *int fseek(FILE *stream, long int offset, int whence)*, gdzie ustawienie *whence* na wartość *SEEK_SET* pozwoli nam na ustalenie absolutnej pozycji. Generalnie bardzo ostrożnie bo raczej nie skaczemy po plikach tekstowych.

Funkcja *ftell()* pozwala nam na sprawdzenie na jakoiej pozycji znajduję się wskaźnik pliku.

3.3.5 Operacje na strumianiach I/O

Generalnie operacji jest mnóstwo i są bardzo rozbudowane, trzeba samodzielnie doczytać szczegóły manuala. Najczęściej używanymi są *fprintf() fscanff()*, ale są też inne. Przykłady:

- *int fgetc(FILE *stream)*
- *int getc(FILE *stream)*
- *int getchar(void)*
- *char * fgets(char *buf, int buflen, FILE *stream)*

- *int fputc(int c, FILE *stream)*
- *int putc(int c, FILE *stream)*
- *int fputs(const char *s, FILE *stream)*
- *int puts(const char *s)*
- *size_t fread(void *ptr,size_t size,size_t nitems,FILE *stream)*
- *size_t fwrite(void *ptr,size_t size,size_t nitems,FILE *stream)*

3.4 Manipulacje strumieniami katalogowymi

Do otwierania strumienia katalogów służy funkcja *opendir()* zwracająca strumień w postaci **DIR***. Do zamykania służy funkcja *closedir()*.

W standardzie POSIX pliki w katalogach są reprezentowane przez specjalne obiekty – **directory entry** zaimplementowane w postaci struktur *dirent*, które zawierają dwa pola: *ino_t d_ino* –numer i-węzła pliku *char d_name[]* – nazwa pliku. Funkcja *readdir()* służy do odczytywania *directory entry* ze strumienia katalogów. Ogólnie potrafi być mało bezpieczna bo w wielowątkowym programie, jeden wątek może nadpisać drugiemu strukturę dirent. Lepiej używać funkcji *readdir_r* – alokuje sam bufor, bierze adres i go wypełnia.

4 Niskopoziomowe operacje wejścia/wyjścia

Interfejs niskopoziomowy jest standaryzowany przec POSIX, a nie C. Umożliwia na wykonanie operacji niskopoziomowych takich jak *read() , write()*. Mamy pewne analogie do funkcji wysokopoziomowych C: zamiast strumieni standardowych: *stdin, stdout, stderr* mamy deskryptory: **STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO**.

Mamy funkcję *fdopen()*, która zwraca nam strumień, więc możemy na nim operować funkcjami wysokopoziomowymi takimi jak *printf() czy scanf()*.

Funkcja *fileno()* zwraca nam deskryptor strumienia pliku

Zamiast *fopen* mamy *open()*, która otwiera istniejący plik w wybranym trybie. Przyjmuję ścieżke pliku w postaci ciągu znaków i **oflag**, która jest sumą logiczną stałych: **O_RDONLY, O_WRONLY, O_RDWR, O_TRUNC, O_NONBLOCK, O_NODELAY, O_APPEND** i zwraca **deskryptor pliku**.

Istotne flagi:

O_CREAT – Tworzy nowy plik jeśli nie istnieje, albo otwiera istniejący

O_EXCL – Tworzy nowy plik, niepowodzenie gdy istnieje

Przykładowy program używający operacji niskopoziomowego wejścia/wyjścia

```
int ret;
char buf[10];
ret = write(STDOUT\_FILENO, "Name? ->", 7);       //odpowiednik printf
ret = read(STDIN\_FILENO, buf, sizeof(buf));     //scanf, obetnie jesli powyzej l
buf[ret] = '\0';          //wazne zeby dodac nullbyte
ret = write(STDOUT\_FILENO, "Hello ", 6);
ret = write(STDOUT\_FILENO, buf, strlen(buf));   //wypisuje bufor
```

5 Cheatsheet funkcji

int fprintf(FILE restrict*stream, const char *restrict format, ..)

void perror(const char *s) - pisze na stderr

int fscanf(FILE restrict*stream, const char *restrict format, ..) – zczytuje ze strumienia według formatu. Zwraca liczbe przypisanych inputów jeśli sukces, EOF jeśli input się kończy przed pierwszą konwersją i bez błędu, EOF i errno jeśli błąd.

char * fgets(char *restrict s, int n, FILE *restrict stream) – zczytuje n-1 bajtów albo do wystąpienia newline i wpisuje jako następny bajt nullbyte. Gdy sukces zwraca s, jeśli strumień jest na EOF ustawia wskaźnik EOF strumienia i zwraca NULL, jeśli błąd ustawia errno i zwraca NULL.

void exit(int status) – kończy proces (EXIT_FAILURE, EXIT_SUCCESS)

int atoi(const char *str) – string to integer. Gdy się nie uda zwraca 0, albo tyle ile da radę. „56test" zwroci 56.

long strotol(const char *restrict str, char **restrict endptr, int base) – Tak samo jak atoi tylko zdefiniowane lepiej I ma endptr (może być null tylko rzutować trzeba np. (char**)NULL) oraz podstawe base.

int getopt(argc, char * const argv[], const char *optstring) – command line parser. Zwraca następną znalezioną opcje gdy znajdzie, ':' gdy znajdzie brakujący wymagany argument, '?' gdy znajdzie opcje nie uwzględnioną w opstringu, -1 gdy wszystko sparsuje.
opstring="t:n::" //t wymagane n opcjonalne
extern char *optarg – przyjmuje wartość aktualnie znalezionego argumentu
extern int opterr – jeśli !=0 będzie wyrzucać błędy na stderr, jeśli ustawimy na 0 getopt będzie tylko zwracać '?' przy błędzie.
extern int optind – indeks następnego elementu argv[] do przetworzenia
extern int optopt – znak opcji, który wywołał error

extern char **environ - trzeba zadeklarować żeby się dostać. Zmienianie: environ[5]

char* getenv(const char *name) – jesli sukces, zwraca pointer do stringa z wartością zmiennej. Jeśli błąd - zwraca NULL

int putenv(char *string – ustawia zmienną środowiskową: "name=value". Jeśli sukces zwraca 0, jeśli błąd - zwraca !=0 i ustawia errno.

int setenv(const char *envname, const char *enval, int overwrite – działa jak putenv tylko podajemy w dwóch argumentach nazwe i wartość. Jeśli overwrite = 0 - nadpisze, overwrite !=0 - nie nadpisze. Jeśli sukces zwraca 0, jeśli błąd - zwraca -1 i ustawia errno.

DIR *opendir(const char *dirname – otwiera strumień katalogu (dla aktualnego "."). Jeśli sukces, zwraca wskaźnik do strumienia, jeśli błąd zwraca NULL i ustawia errno.

DIR *fdopendir(int fd) – działa jak opendir, ale przyjmuje file descriptor.

int closedir(DIR *dirp) – Zamyka strumień. Sukces: 0, Błąd: -1 i errno.

struct dirent * readdir(DIR *dirp) – czyta po kolei struktury **dirent** dla plików w katalogu. Nie zwraca dla plików z pustymi nazwami. Sukces: wskaźnik do struktury lub NULL gdy koniec, ale nie ustawia errno, Błąd: NULL i errno

struct dirent – Zawiera: ino_t d_ino - Numer seryjny pliku, char d_name[] - nazwa pliku.

struct stat – struktura na dokładne informacje o pliku. Wszystkie pola w man 2 lstat. mode_t st_mode - pole opisujące typ i tryb pliku. Makra do odczytywania st_mode w man 7 inode. Główne: S_ISREG, S_ISDIR, S_ISLINK.

int stat(const char *restrict path, struct stat *restrict buf – wpisuje informacje o pliku do struktury stat. Sukces: 0, Błąd: -1 i errno.

int lstat(const char *restrict path, struct stat *restrict buf) – działa jak stat, ale dla symlinków daje info o symlinkach a nie o pliku do którego się odnoszą.

char *getcwd(char *buf, size_t size) – wstawia ścieżke current working directory do miejsca wskazanego przez buf, size: długość tablicy wskazywanej przez buf (gdy NULL unspecified zachowanie). Sukces: zwraca buf, Błąd: NULL i errno.

int chdir(const char *path) – ustawia CWD na wartość wskazywaną przez path. Sukces 0, Błąd: -1 i errno.

int nftw(const char *path, int (*fn)(const char*, const struct stat *, int, struct FTW *), int fd_limit, int flags) – przechodzi w dół drzewa plików. Argumenty:

- path – ścieżka od której zaczynamy iść w dół
- fn – wskaźnik do funkcji opisanej przez nas w powyższy sposób. nftw wykonuje ją na każdym pliku i przypisuje po kolei: ścieżkę pliku, strukture stat o pliku, int type do którego mamy stałe - mówi nam o typie pliku, strukture FTW - chyba flaga jaką mamy ustawioną.
- fd_limit – maksymalna liczba użytych deskryptorów plików
- flags – użyte flagi: FTW_PHYS - nie wchodzi w głąb symlinków.

Koniecznie #define _XOPEN_SOURCE 500 ,przed wszystkim innym, bez tego nie znajdzie nftw. Zwraca: 0 - drzewo się skończyło, -1 i errno - błąd, coś innego - nasze fn zwróciła !=0 wtedy zwraca te wartość.

int ftw(const char *path, int (*fn)(const char*, const struct stat *ptr, int flag), int ndirs) –

FILE *fopen(const char* restrict path, const char* restrict mode) - otwiera strumień. tryb: r-readonly, w-obcina do zero albo tworzy nowy writeonly, a-append otwiera w punkcie EOF, r+ - read and write, w+ obcina do zera i write and read, a+- read and append. Sukces: zwraca pointer na obiekt strumienia, Błąd: null i errno.

int fclose(FILE *stream) – Zamyka strumień. Sukces: 0, Błąd: EOF i errno.

int fseek(FILES *stream, long offset, int whence) - Przesuwa wskaźnik strumienia o offset od pozycji SEEK_SET - początkowa, SEEK_CUR - aktualna, SEEK_END - eof. Sukces: 0, Bład: -1 i errno.

int unlink(const char *path) – usuwa directory entry powiązane z plikiem (czyli w sumie chyba plik). Sukces: 0, Błąd: -1 i errno. UWAGA: errno ma wartość ENOENT gdy się nie udało bo plik nie istniał.

mode_t umask(mode_t cmask) – ustawia aktualną umaske (umaska jest w systemie ósemkowym). Ex. umask(permissions&0777). Zwraca starą umaske.