Notatki Systemy Operacyjne

Piotr Kotynia

25 kwietnia 2022

Notatki studenckie zrealizowane na podstawie wykładów mgr inż. Pawła Sobótki uzupełnione o wiedzę ze slajdów, manuala, internetu i własną interpretację na przedmiot Systemy Operacyjne 1. Większa część notatek to opisy teorytyczne bez dokładnych opisów funkcji. W tym celu stworzyłem na końcu cheatsheet, który również nie jest kompletnym opisem zachowania funkcji, ale może pomóc w przypomnieniu lub wstępnym zrozumieniu funkcji lub struktury. Ostatecznie żeby zrozumieć dokładnie funkcję lub mechanizm i tak polecam przeczytać manuala. Notatki mogą być niekompletne, potencjalnie zawierać błędy i niepoprawne uproszczenia.

Spis treści

I Systemy Operacyjne 1 **4**

1 Systemy operacyjne i komputerowe - wstęp **4**

1.1 System operacyjny, a system komputerowy 4

1.1.1 Co to jest system operacyjny? 4

1.1.2 Składowe systemu komputerowego 5

1.1.3 Tryby pracy systemu komputerowego 5

1.2 Zadania systemów operacyjnych 6

1.3 Działanie systemu komputerowego 7

1.3.1 Przerwania 7

1.3.2 Obsługa wejścia/wyjścia 8

2 Procesy **8**

2.1 Koncepcja procesu 8

2.1.1 Składowe procesu 8

2.1.2 Stan procesu 8

2.1.3 Blok kontrolny procesu (PCB) 9

2.1.4 Przełączanie procesora między procesami 9

2.2 Planowanie procesów 10

2.2.1 Kolejki planowania procesów 10

2.3 Działania na procesach 11

2.3.1 Tworzenie procesu 11

2.3.2 Planiści (schedulery) 12

2.3.3 Kończenie procesu 12

2.4 Środowisko wykonania procesu 13

3 Interfejs systemu plików, strumieniowe wejście/wyjście **14**

3.1 Koncepcja pliku 14

3.1.1 Operacje plikowe 14

3.1.2 Blokady dostępu do plików 15

3.1.3 Otwarte pliki 15

3.2 Struktura katalogowa plików 16

3.2.1 Rodzaje struktur katalogowych 16

3.2.2 Katalog o strukturze acyklicznego grafu 16

3.2.3 Montowanie podsystemu plików 16

3.2.4 Ochrona plików 17

3.3 Input/Output 17
3.3.1 Strumieniowe wejście/wyjście 17
3.3.2 Buforowanie strumieni 18
3.3.3 Blokowanie strumieni, EOF i błędy 19
3.3.4 Pozycja strumienia 19
3.3.5 Operacje na strumianiach I/O 19
3.4 Manipulacje strumieniami katalogowymi 20

4 Niskopoziomowe operacje wejścia/wyjścia 20

4.1 O_NONBLOCK i pliki FIFO 21
4.2 Struktury danych operacji I/O 22
4.3 Synchroniczne operacje na plikach (do napisania) 23

5 Sygnały POSIX 23

5.1 Obsługa sygnałów 23
5.2 Własne procedury obsługi sygnałów 24
5.3 Blokowanie sygnałów 24

6 Wątki i Muteksy 24

6.1 Podstawy wielowątkowości 24
6.2 Mutexy 25
6.3 Anulowanie wątków 26
6.4 Cleanery 26

II Systemy Operacyjne 2 26

7 Komunikacja międzyprocesowa (IPC) 26

7.1 Rodzaje komunikacji 26
7.2 Łącza 28
7.2.1 Łącza anonimowe - pipe 28
7.2.2 Łącza nazwane - FIFO 28
7.2.3 Błędy i SIGPIPE 29
7.3 Kolejki komunikatów 29
7.4 Pamięć dzielona 30
7.4.1 Odwzorowanie plików w pamięci 31
7.4.2 Współdzielone segmenty pamięci 31
7.4.3 System V 31

8 Synchronizacja **31**

9 Interfejs gniazd **32**

10 Cheatsheet funkcji i struktur **32**

Część I

Systemy Operacyjne 1

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).

1.2 Zadania systemów operacyjnych

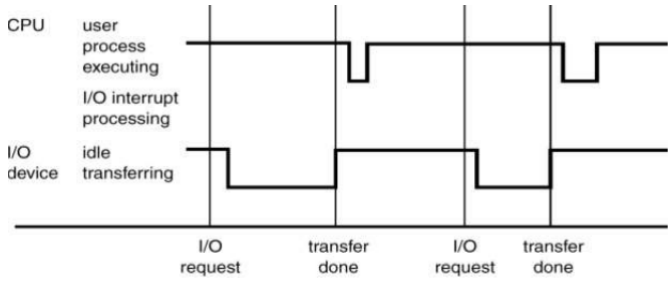
Efektywne zarządzanie zasobami systemu komputerowego

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



Rysunek 1: Uproszczony model obsługi przerwania przez CPU

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.

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

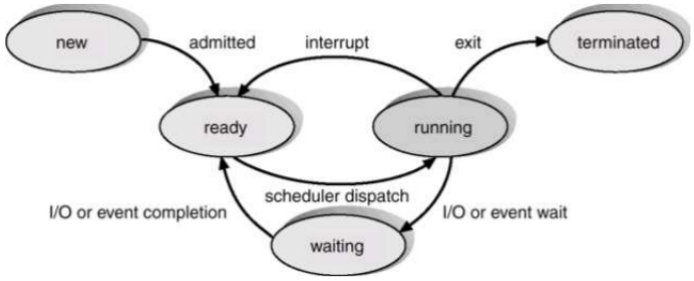
Proces – (zadanie) wykonujący się program. Procesy są rozróżniane za pomocą identyfikatorów procesów **(PID - process identifier)**, które są liczbami całkowitymi. Jeden program może tworzyć wiele procesów.

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



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)

- Oczekujący (waiting) – proces czeka na wystąpienie jakiegoś zdarzenia
- Gotowy (ready) – proces czeka na przydział procesora
- Zakończony (terminated) – proces zakończył działanie

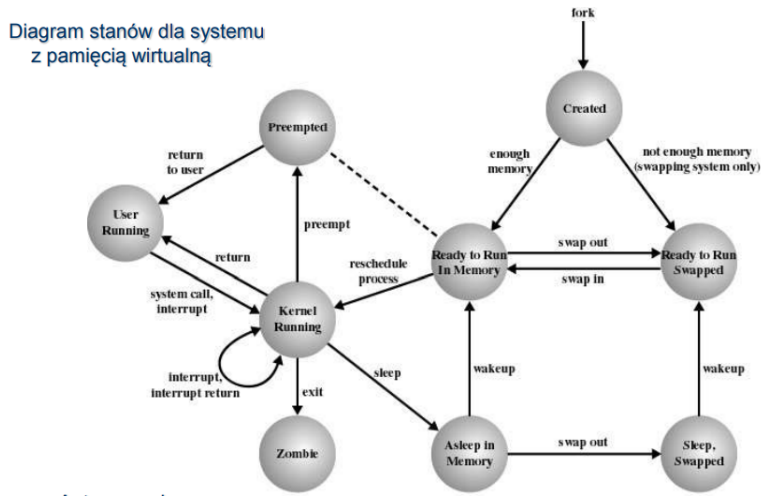
2.1.3 Blok kontrolny procesu (PCB)

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):

- 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:



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

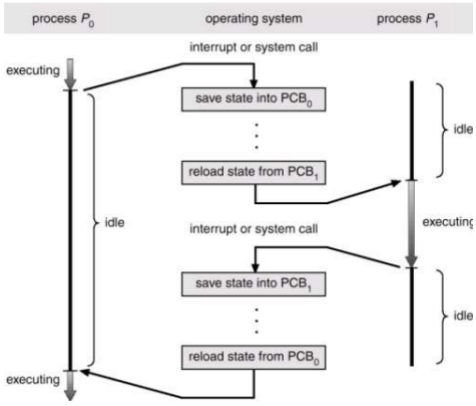
- przerwanie zegarowe
- przerwania od urządzeń
- wywołanie f. systemowej
- wystąpienie pułapki

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



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

2.3 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] W niektórych systemach zamiast procesu init występuje proces **systemd**

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

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]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

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
- 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()* słuzy 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 przez 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**.

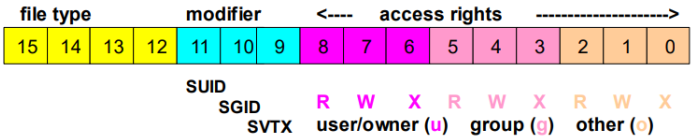
Istotne flagi:

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

O_EXCL – Tworzy nowy plik, niepowodzenie gdy istnieje

Zamiast funkcji *fseek()* mamy funkcje *lseek()*, zamiast *fclose()* mamy *close()*.

Przykładowy program używający operacji niskopoziomowego wejścia/wyj-



File attribute	Value (octal)	Value (symbolic)
FIFO	0010000	S_IFIFO
Special character device	0020000	S_IFCHR
Directory	0040000	S_IFDIR
Special block device	0050000	S_IFBLK
Ordinary file	0100000	S_IFREG
Symbolic link	0120000	S_IFLNK
Socket	0140000	S_IFSOCK

Rysunek 5: Diagram atrybutów plików: trochę nie czaje tabelki

ścia

```
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
```

4.1 O_NONBLOCK i pliki FIFO

Typ FIFO, inaczej plik łącza nazwanego, może być otwarty do zapisu i odczytu, proces który otwiera go do zapisu "wkłada" do niego bajty jak do kolejki, a z drugiego końca inny proces konsumuje te bajty przez czytanie. Umożliwia on komunikacje między procesami. Do stworzenia takiego pliku można użyć polecenia *mkfifo*. Flaga O_NONBLOCK służy głównie do otwierania plików FIFO.

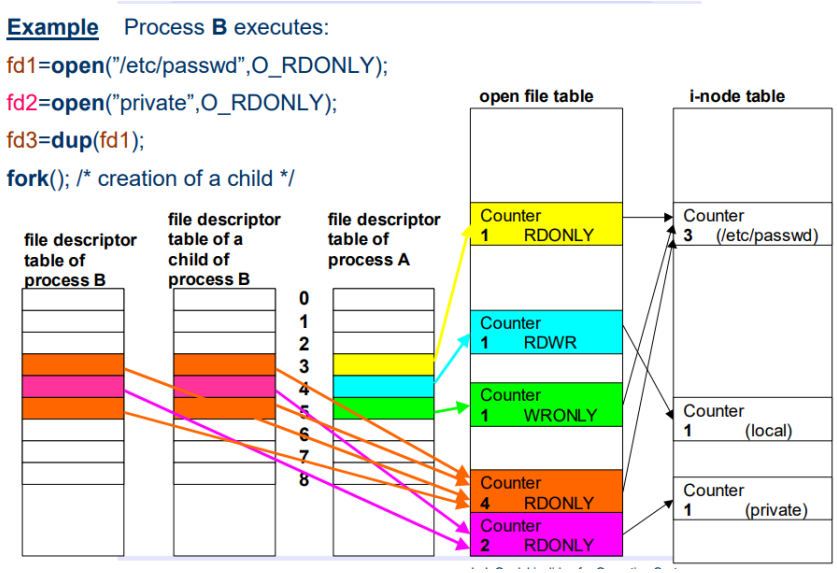
Jeżeli nie jest ustawiona, czyli plik jest w trybie blokującym: gdy otwieramy plik do odczytu, open() zablokuje aktualny potok i będzie czekał aż jakiś wątek otworzy plik do zapisu. Analogicznie tryb do zapisu będzie czekał aż plik zostanie otworzony z drugiej strony do odczytu.

W trybie nieblokującym: jeśli otworzymy do odczytu, open() powinien się wykonywać bez czekania i np. read() będzie się wykonywał nawet gdy nie ma

czego odczytać. Jeśli otworzymy plik jedynie do zapisu, open() zwróci błąd jeśli żaden proces nie ma otwartego pliku do odczytu.

4.2 Struktury danych operacji I/O

Każdy proces ma swoją **tablice deskryptorów plików**, ale oprócz tego istnieje **tablica otwartych plików** kernela. Deskryptory plików w tablicy procesów są jedynie wskazaniami na miejsca w tablicy otwartych plików jądra. Z kolei wpisy w tablicy jądra są referencjami do **tablicy i-node'ów** w systemie plików.



Rysunek 6: Co się dzieje? Każde otwarcie pliku, nawet jeśli odnosi się do tej samej ścieżki, jest nowym wpisem w tablicy otwartych plików. Aby 2 deskryptory wskazywały na to samo miejsce w tablicy należy użyć funkcji *dup()*, albo stworzyć nowy proces funkcją *fork()*.

Dlaczeg to jest istotne? Jeżeli proces otworzy kilka razy ten sam plik, deskryptory będą wskazywać na rózne elementy tablicy otwartych plików, czyli nie będą np. dzielić wskaźnika gdzie aktualnie znajdujemy się podczas czytania pliku. Jeśli chcemy aby deskryptory wskazywały na ten element tablicy, należy użyć funkcji *dup*, albo *dup2()* Proces dziecko dziedziczy deskryptory otwartych plików od procesu rodzica, czyli wskazują na to samo miejsce w tablicy otwartych plików.

4.3 Synchroniczne operacje na plikach (do napisania)

5 Sygnały POSIX

Sygnał – mechanizm za pomocą którego proces lub wątek może się dowiedzieć o zdarzeniu, które wystąpiło w systemie. Do wysyłania sygnałów służy funkcja *kill()*. Długa i szczegółowa lista sygnałów i ich opis znajduje się pod man 7 signal.

Nr	Name	Meaning	Action
1	**SIGHUP**	Hangup	Exit
2	**SIGINT**	tty interrupt (typically: ^C)	Exit
9	**SIGKILL**	**Unconditional** process termination	Exit
11	**SIGSEGV**	Segmentation Fault	Core dump + exit
13	**SIGPIPE**	Broken Pipe	Exit
14	**SIGALRM**	Alarm Clock	Exit
15	**SIGTERM**	Software interrupt	Exit
	SIGUSR1,2	Two „user interrupts" (no pre-defining meaning)	Exit
	SIGCHLD	Child Status Changed	Ignore
	SIGCONT	Process to be continued	Continue
	SIGSTOP	**Unconditional** stop for a process	Stop
	SIGTSTP	Stop of a process via tty (typically: ^Z)	Stop
	SIGTTIN	Stopped (tty input)	Stop
	SIGTTOU	Stopped (tty output)	Stop

Rysunek 7: Lista najważniejszych sygnałów

Sygnały mogą być dostarczone do procesu albo do wątku (wątki będą później).

5.1 Obsługa sygnałów

Każdy sygnał ma zawsze zdefiniowany sposób w jaki zachowa się w odpowiedzi na każdy sygnał. Sygnał może być:

dostarczony (delivered) – jeśli odpowiednia akcja procesu jest wywołana: ignorowanie lub wywołanie **obsługi sygnału (signal handlers)** zdefiniowanej przez użytkownika albo domyślnej.

zaakceptowany (accepted) – jawne zaakceptowanie sygnału, np. kiedy sygnał jest zwrócony przez sigwait() - proces aktywnie czekał na sygnał, czyli sygnał został obsłużony **synchronicznie**.

zablokowany (blocked) – proces może zażyczyć sobie żeby sygnały, które otrzymuje oczekiwały na dostarczenie.

POSIX nie daje żadnej gwarancji co do tego ile razy proces otrzyma dany sygnał. Wysłany kilkukrotnie sygnał "skleja" się w jeden sygnał jeśli zostały wysłane w krótkim czasie. Nie jest zdefiniowana również kolejność w jakiej kilka różnych sygnałów zostanie obsłużona.

5.2 Własne procedury obsługi sygnałów

Funkcja do osbługi sygnału musi mieć następującą sygnature: *void handler_name(int signo)*. Mając taką funkcje możemy korzystać z metody API *sigaction()*, aby zdefiniować nową procedurę obsługi sygnału. Nie można przedefiniować procedury obsługi sygnału **SIGKILL**.

Proces potomny dziedziczy procedury obsługi sygnału.

5.3 Blokowanie sygnałów

Możemy w programie definiować, które sygnały będą przez proces blokowane. Struktura sygnałów, które będziemy blokować nazywa się *sigset_t*. Do zerowania tej struktury używamy *sigemptyset()*, dodajemy do zbioru sygnałów funkcją *sigaddset()* i konfigurujemy sygnały blokowane funkcją *sigprocmask()*.

6 Wątki i Muteksy

6.1 Podstawy wielowątkowości

Każdy proces ma dostęp do zasobów wymaganych żeby wykonać swoje zadanie. Może w tym celu użyć jednego lub więcej wątków. Dzielą one dostęp do wspólnych zasobów procesu, ale mogą też posiadać własne zasoby. Do zalet wielowątkowości należy:

- responsywność – proces może się wykonywać nawet jeżeli jego część się zablokuje.

- dzielenie zasobów procesów – wątki dzielą zasoby procesy, to łatwiejsze niż współdzielona pamięć i przekazywanie wiadomości
- ekonomia – taniej jest przełączyć kontekst niż utworzyć nowy proces
- skalowalność – proces może wykorzystać wieloprocesorową architekturę aby wykonywać wątki współbieżnie

Wątki dzielą się na wątki **jądra**(WIndows, Solaris, Linux, itp.) i **użytkownika**(POSIX threads-Pthreads, Windows Threads, Java Threads). Itnieją różne modele mapujące wątki użytkownika na wątki jądra w sposób: many-1, 1-1 lub many-many.

My korzystamy z biblioteki *Pthreads* do zarządzania wątkami. Istnieje wiele problemów związanych z wprowadzeniem wielowątkowości, np. czy funkcja *fork()* powinna duplikować dany wątek czy wszystkie wątki? (POSIX definiuje, że powinniśmy duplikować tylko wątek wywołujący).

Każdy wątek ma swój własny prywatny stos. Wątki współdzielą przestrzeń adresową więc mają dostęp do swoich stosów nawzajem.

6.2 Mutexy

Mutex – obiekt synchronizujący, którego celem jest umożliwienie wielu wątkom serializować ich dostęp do współdzielonych danych. Nazwa pochodzi od mutual-exclusion – wzajemne wykluczanie. Wątek blokuje mutex i staje się jego właścicielem dopóki sam go nie odblokuje. Do blokowania służą funkcje: *pthread_mutex_lock()*(tryb blokujący) i *pthread_mutex_trylock()*(tryb nieblokujący). Do odblokowania: *pthread_mutex_unlock()*.

Mutex należy najpierw zainicjalizować funkcją *pthread_mutex_init()*, która wymaga również wskaźnika do struktury zawierającej atrybuty mutexu (ją również należy najpierw zainicjować specjalną funkcją). Póki co nie korzystamy z zaawansowanych funkcji więc "ustawiamy NULL i jest fajnie".

Mutex można też zainicjalizować specjalnym makrem mutex = PTHREAD_MUTEX_INITIALIZER.

Mutexów pod żadnym pozorem nie można przypisywać/kopiować!

Nieużywany mutex niszczymy funkcją *pthread_mutex_destroy()*

6.3 Anulowanie wątków

Anulowanie wątków to mechanizm, który pozwala jednemu wątkowi zakończyć wykonanie dowolnego innego. Każdy wątek ma skojarzone 2 specjalne atrybuty: **cancelability** – mówi nam o tym czy wogóle można anulować wątek oraz **cancelability type** – mówi o tym w jaki sposób nastąpi anulowanie. Do ustawiania tych atrybutów mamy funkcje pthread_setcancelstate() i pthread_setcanceltype().

- PTHREAD_CANCEL_DEFERRED - domyślna wartość, oznacza że wątek zakończy się przy anulowaniu dopiero gdy będzie w trakcie wykonywania jednej z funkcji, która znajduje się w liście *cancellation points*.
- PTHREAD_CANCEL_ASYNCHRONOUS – wątek zakończy się od razu po anulowaniu, być może nawet w trakcie wykonywania jakiejś funkcji maszynowej.

Funkcja pomocnicza, która po prostu jest cancellation point to *pthread_testcancel()*.

Jeśli przerwiemy funkcje i przez to nie zwróci ona wartości to zwrócony wskaźnik będzie miał wartość PTHREAD_CANCELED.

6.4 Cleanery

Przy anulowaniu wątków czasem chcemy aby zanim funkcja wyjdzie wykonała jakąś operacje. Służą do tego funkcje *pthread_cleanup_push()* i *pthread_cleanup_pop()*. Muszą one znajdować się na równym poziomie, pierwsza służy do ustawienia funkcji, która ma się wykonać przy anulowaniu, druga ustawia miejsce, w którym usuwamy ją z listy do wykonania i wykonujemy ją lub nie.

Część II

Systemy Operacyjne 2

7 Komunikacja międzyprocesowa (IPC)

7.1 Rodzaje komunikacji

Procesy dzielą się na dwa rodzaje ze względu na interakcje z innymi procesami:

- Procesy niezależne – nie mogą oddziaływać na inne procesy inne procesy nie mogą oddziaływać na nie.
- Procesy współpracujące – oddziałują wzajemnie na swoje wykonanie. System udostępnia takim procesom: współbieżne wykonanie oraz usługi synchronizacji i komunikacji.

Procesy współpracujące dzielą się ze względu na model komunikacji:

- Pamięć wspólna:
  1. Komunikacja pod kontrolą użytkownika
  2. Największa szybkość i oszczędność pamięciową komunikacji
  3. Problematyczna synchronizacja dostępu dodanych
  4. Zjawisko **data race** – dwa procesy "ścigają się" w dostępie do tego samego obszaru pamięci powodując liczne problemy np. nadpisując sobie nawzajem zmienne
- Przekazywanie komunikatów
  1. Komunikacja pod jądra systemu
  2. Prosta synchronizacja w realizowana przez system
  3. Brak problemów związanych z współdzieleniem tych samych obszarów pamięci
  4. Kopiowanie danych zmniejsza efektywność

Metody komunikacji międzyprocesowej (IPC):

- Sygnały
- Pliki współdzielone
- Muteksy
- Kod wyjścia procesu
- Strumienie standardowe
- Łącza zwykłe i nazwane (FIFO)
- Zestawy interfejsów POSIX np. UNIX System V IPC
- Gniazda (sockets)

7.2 Łącza

Łącze tworzy kanał komunikacji między dwoma procesami. Istnieją dwa rodzaje łączy:

- **Łącza zwykłe(anonimowe)** – Reprezentowane przez **pipe**. Po utworzeniu nie są widoczne przez inne procesy, ale dostęp do nich może być przekazany.
- **Łącza nazwane** – Reprezentowane przez **FIFO** mogą być udostępniane każdemu procesowi dzięki nazwie.

Przekazywanie danych jest na zasadzie producent-konsument. Producent pisze dane do jednego końca łącza, a konsument czyta je z drugiego.

7.2.1 Łącza anonimowe - pipe

Cechą łącza anonimowego jest to, że **nie jest widoczny w systemie plików**. Do utworzenia służy funkcja *pipe()*. Przyjmuje ona tablice dwóch integerów, i zapisuje do pierwszego deskryptor służący do odczytu, a do drugiego deskryptor do zapisu. Trzeba też jednak przekazać te deskryptory procesowi z którym chcemy się komunikować np. poprzez *fork()* – wtedy dziecko odziedziczy tablice deskryptorów, lub przez gniazda lokalne (później).

Mamy też wysokopoziomowy interfejs otwierania pipe, zwracający nam plik **FILE*. Służy do niego funkcja *popen()*. Cechy łącza anonimowego:

- Dostępne tylko poprzez deskryptory
- Brak wsparcia pozycjonowania – odczyt i zapis bez ustawiania pozycji ze skutkiem błędu **ESPIPE**
- Nie można pisać do zamkniętego łącza
- Dane w łączu to zlepek bajtów bez logicznego odseparowania
- Odczyt i zapis jest nierozdzielny (atomic) dla danych nie większych od **PIPE_BUF** czyli maksymalnej pojemności łącza. Gdy przekroczymy ten limit proces czeka aż inny proces nie odczyta tych bajtów z drugiej strony.

7.2.2 Łącza nazwane - FIFO

Co do zasady działania FIFO nie różni się od łącza anonimowego, za to jest widoczny w systemie plików. Do utworzenia FIFO służy funkcja *mkfifo()*. Przyjmująca ścieżkę i tryb praw dostępu do pliku.

Domyślnie otwieranie jest blokujące, czyli otwarte FIFO czeka na otworzenie drugiego końca. Można otworzyć nieblokująco koniec do odczytu, ale pod żadnym pozorem nie do zapisu, nie można pisać do pliku, z którego nic nie czyta.

7.2.3 Błędy i SIGPIPE

Prawidłowa obsługa błędów i sygnałów w przypadku łącz może być problematyczna, i trzeba wiedzieć o kilku rzeczach. Próba zapisu do zamkniętego łącza ustawia nam błąd **EPIPE** i wysyła do procesu sygnał **SIGPIPE**, który domyślnie zabija proces.

W przypadku nieblokującego zapisu danych rozmiaru co najwyżej **PIPE_BUF**: jeśli mamy wystarczająco dużo miejsca zapis się uda, jeśli nie to nie zapiszemy żadnych danych w łączu i otrzymamy błąd **EAGAIN**.

W przypadku nieblokującego zapisu danych większych od PIPE_BUF: jeśli przynajmniej jeden bajt może być zapisany - zapis się uda, jeśli nie to nie zapiszemy żadnych danych w łączu i otrzymamy **EAGAIN**.

W przypadku odczytu dla pustego łącza:

Jeśli żaden proces nie ma otwartego końca do zapisu, funkcja read() w obu trybach: blokującym i nieblokującym zwraca 0, czyli end-of-file.

Jeśli jakiś proces ma otwarty koniec do zapisu: w trybie blokującym wątek czeka aż pojawią się dane, w trybie nieblokującym *read()* zwraca -1 i ustawia errno na **EAGAIN**.

7.3 Kolejki komunikatów

Kolejki komunikatów rozwiązują problem, który ma łącze nienazwane w postaci *pipe*, czyli sklejania się wiadomości. Składają się z komunikatów, którym możemy nadać różne priorytety. Niezwiązane procesy mogą też swobodnie dostawać się do tej samej kolejki,

Kolejki mają trwałość w ramach systemu, czyli żyją do jego restartu lub jawnego usunięcia. W przypadku Linuxa kolejka jest widoczna w systemie plików, jest identyfikowana przez deskryptor kolejki (może być implementowany jako deskryptor pliku) i znajduje się ona w specjalnej, dedykowanej dla kolejek przestrzeni nazw. Wirtualne pliki kolejek są w folderze */dev/mqueue/* (katalog ma płaską strukture). Nie są one fizycznie obecne na dysku, ale są widoczne jako pliki analogicznie do procesów w katalogu */proc*. Atrybuty kolejki określa struktura **mq_attr** zawierająca:

- **mq_maxmsg** – maksymalna liczba wiadomości w kolejce, gdy przekroczona proces piszący się zablokuje
- **mq_msgsize** – maksymalna długość wiadomości
- **mq_flags** – 0 albo NON_BLOCK
- **mq_curmsgs** – aktualna liczba komunikatów

Przekazywanie komunikatów wielkości mniejszej niż max. dla kolejki jest niezawodne. Deskryptor kolejki jest zmienną typu **mqd_t** (czyli w sumie int).

Implementacja defniniuje też stałe określające cechy kolejki

- **MQ_PRIO_MAX** – maksymalny możliwy priorytet
- **MQ_OPEN_MAX** – maksymalna liczba kolejek otwartych przez jeden proces.

Parametry kolejki można zmieniać w */proc/sys/fs/mqueue*.

Do tworzenia/otwierania kolejki służy *mq_open()*, do zamykania *mq_close()* i kasowania *mq_unlink()*.

Do wysyłania komunikatów: *mq_send()* i *mq_timedsend()*, a do odbierania: *mq_receive()* i *mq_timedreceive()*.

Wersje z *timed* służą do ograniczenia czekania.

Do pobierania i ustawiania atrybutów kolejek służą: *mq_getattr()* i *mq_setattr()*.

Możemy też dla procesu ustawić opcję powiadamiania procesu o pojawieniu się wiadomości w konkretnej kolejce. Do wyboru jest powiadamianie przez sygnał lub przez wywołanie wątku, działa to analagicznie do asynchronicznego API I/O.

Służy do tego funkcja *mq_notify()*. W danym momencie tylko jeden proces może zasubskrybować powiadomienie dla konkretnej kolejki. Próba rejestracji powiadomienia gdy inny proces już to zrobił powinna się nie udać. Po każdym powiadomieniu trzeba odnowić subskrypcje ustawiając powiadomienie ponownie. Powiadomienie jest wywoływane tylko w momencie gdy kolejka jest pusta i nagle dostanie wiadomość.

7.4 Pamięć dzielona

Procesy moga się ze sobą komunikować poprzez dzielenie tej samej przestrzeni adresowej. Jest to trudne ze względów bezpieczeństwa i komunikacji, ale również

bardzo szybkie.

7.4.1 Odwzorowanie plików w pamięci

Jednym ze sposobów jest bezpośrednie odwzorowanie części pliku w przestrzeń adresową procesu. Należy najpierw plik otworzyć, a następnie zmapować przestrzeń funkcją *mmap()*. Otrzymujemy w ten sposób wskaźnik, który odwołuje sie do zmapowanej pamięci. Następnie możemy tłumaczyć operacje wykonane na pliku w przestrzeni adresowej procesu na operacje dyskowe. Funkcja*mmap()* to bardzo obszerne narzędzie i my będziemy korzystać tylko z części jej możliwości.

Możem zmapować pamięć w taki sposób żeby zmiany były widoczne dla innych procesów lub uczynić tak żeby proces miał swoją kopie pamięci pliku i zmiany będą dokonywane tylko na potrzeby danego procesu.

7.4.2 Współdzielone segmenty pamięci

Procesy mogą również tworzyć segmenty pamięci, do których dostęp będą miały inne procesy. Funkcją *shm_open()* tworzymy połączenie między segmentem pamięci, a deskryptorem pliku. Otrzymany deskryptor, służy nam do odwzorowania segmentu na przestrzeń adresową procesu funkcją *mmap()* tak jakby to był zwykły plik. Każdy proces, który ma nazwę segmentu (i własciwe prawa) jest w stanie otworzyć go i zmapować w swojej przestrzeni adresowej.

Po otworzeniu segmentu należy określić jego rozmiar funkcją *ftruncate()* i określić odwzorować segment na przestrzeń adresową procesu funkcją *mmap()* z flagą **MAP_SHARED**. Po zakończeniu usunąć odwzorowanie funkcją *munmap()*.

7.4.3 System V

TODO

8 Synchronizacja

TODO

9 Interfejs gniazd

Gniazda reprezentują punkt końcowy komunikacji. Każde gniazdo ma typ i określony protokół. Są one dostępne przez deskryptor przydzielany przez system przy jego tworzeniu. Gniazda w domenie lokalnej (UNIX) są widoczne w systemie plików jako pliki specjalne typu **socket** i wykorzystuje się je do komunikacji międzyprocesowej w obrębie tego systemu. Można do nich pisać i z nich czytać tak jak to wygląda w normalnych plikach lub łączach.

W przeciwieństwie do łączy, gniazda wspierają komunikacje pomiędzy niepowiązanymi procesami, a nawet pomiędzy procesami działającymi na różnych urządzeniach, komunikujących się poprzez sieć!

Tworząc łącze musimy określić sposób komunikacji: jaka będzie jednostka transmisji?, czy dane mogą zostać utracone w trakcie komunkacji?, jak wiele socketów będzie brało udział w komunikacji?.

10 Cheatsheet funkcji i struktur

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. W stat nie ma informacji o nazwie pliku! - plik sam w sobie nie wie jaką ma nazwe, przecież może mieć kilka nazw.

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.

int fstat(int fd, struct stat *buf) – działa jak stat, ale przyjmuje deskryptor pliku.

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.

int open(const char *pathname, int oflag, mode_t mode) – niskopoziomowa funkcja do otwierania istniejącego lub nieistniejącego pliku. Argumenty:

- pathname - ścieżka do pliku

- oflag - suma logiczna flag (pełna lista w manie)
- mode - prawa przy tworzeniu pliku (RWX-RWX-RWX). Pamiętając że efektywnie i tak będzie to zależało od umaski: mode & umask. Pierwsza trójka bitów to sticky bity (SUID-SGID-SVTX). Jeśli nie ustawione to dziedziczy po procesie wywołującym.
    1. SUID – Jeśli ustawiony to plik otwierany jest z uprawnieniami użytkownika (UID użytkownika staje się tymczasowo UID'em właściciela), który jest właścicielem pliku (np. passwd musi mieć ten bit bo innaczej użytkownicy nie mogliby edytować swojego hasła)
    2. SGID – Działa podobnie do SUID tylko odnosi się do grupy. Otwieramy plik tak jakbyśmy byli członkami grupy do której plik należy.
    3. SVTX – Tak zwany Sticky Bit. Oznacza, że na plik (albo katalog) może być usunięty lub może mu być zmieniona nazwa tylko przez właściciela pliku (albo roota oczywiście). Czyli nawet mając wszystkie uprawnienia do pliku, inny użytkownik nie może usunąć pliku innemu użytkownikowi. Przydatne np. dla katalogu /tmp aby użytkownicy nie usuwali sobie nawzajem plików.

Sukces: otwiera plik zwraca najmniejszy, nieużywany, nieujemny deskryptor pliku. Porażka: Zwraca -1, nie tworzy żadnego pliku i ustawia errno.

ssize_t read(int fd, void *buf, size_t nbyte) – funkcja próbuje przeczytać *nbyte* bajtów z pliku powiązanego z deskryptorem pliku *fd* do bufora *buf*. Zachowanie dla kilku readów z tego samego pipe, FIFO albo terminala jest niezdefiniowane.

int kill(pid_t pid, int sig) – funkcja wysyła sygnał **sig>0** do procesu. W zależności od pid funkcja wysyła sygnał do:

- pid>0 – Do procesu z danym PID
- pid==0 – Do wszystkich procesów które należą do grupy procesów wysyłającego procesu. Zwykle do wszystkich dzieci i mozliwe kilku przodków.
- pid==-1 – Do wszystkich procesów w systemie (oprócz **init**)
- pid<-1 – Do wszystkich procesów, które należą do grupy procesów **pgid==-pid**

Dla sig==0, sygnał nie jest wysyłany, ale standardowe sprawdzanie błędów jest wykonywane. Sukces: zwraca 0. Porażka: zwraca -1 i ustawia errno.

int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact) – funkcja służy do konfigurowania nowej procedury obsługi sygnału lub/i sprawdzenia starej.

- sig – numer sygnału, którego osbługę chcemy zmienić
- act – wskaźnik do struktury zawierającej nową konfiguracje obsługi sygnału.
- oact – wskaźnik do struktury do której zapisana zostanie stara procedura.

Sukces: 0. Porażka: -1 i errno.

struct sigaction – struktura zawierająca konfiguracje obsługi sygnału, zawiera:

- void(*sa_handler) (int) sa_handler – wskaźnik do funkcji obsługującej sygnał: funkcja musi przyjmować int i zwracać void. Istnieją makra SIG_DFL (domyślnie) i SIG_IGN (ignoruj sygnał).
- sigset_t sa_mask – maska sygnałów które będą blokowane podczas obsługi sygnałów
- int sa_flags – flagi zmieniające zachowanie sygnałów (pełna lista man 3p sigaction)
- void(*) (int, siginfo_t *, void *) sa_sigaction – kolejny wskaźnik do funkcji obsługującej sygnał ale bardziej rozbudowana - powinno się używać tylko jednej z dwóch.

Sukces: 0. Porażka -1 i errno.

int sigemptyset(sigset_t *set) – inicjalizuje pusty set sygnałów w miejsce wskazane przez *set*. Sukces: 0. Porażka: -1 i errno.

int sigaddset(sigset_t *set, int signo) – Dodaje sygnał o numerze *signo* do struktury *set*. Wcześniej należy zainicjować pusty set funkcją powyżej. Sukces: 0. Porażka: -1 i errno.

int sigprocmask(int how, const sigset_t *restrict set, sigset_t restrict oset) – sprawdza i/lub zmienia maskę sygnałów blokowanych.

- how – ustala w jaki sposób zmieniamy maskę, musimy wybrać jedną z możliwości:
  1. SIG_BLOCK – nowa maska będzie sumą aktualnego i nowego zestawu (setu)
  2. SIG_SETMASK – nowa maska zastąpi starą
  3. SIG_UNBLOCK – nowa maska będzie częścią wspólną aktualnego i nowego zestawu
- set – nowy set sygnałów, jeśli null to możemy w ten sposób sprawdzić aktualnie blokowane sygnały
- oset – miejsce gdzie zapiszemy starą maske

Jeśli są jakieś oczekujące niezablokowane sygnały po użyciu funkcji, przynajmniej jeden powinien został dostarczony zanim funkcja zwróci wartość.

Sukces: 0. Porażka: -1 i errno.

pid_t wait(int *stat_loc) – funkcja wait służy do uzyskiwania informacji od procesów dzieci - wszystkich. Przy wywołaniu wątek blokuje się dopóki nie uzyska informacji o zakończeniu procesu dziecka, lub nie dostanie sygnału który każe mu zrobić coś innego, może również wystąpić błąd. Jeśli wystąpi zakończenie dwóch lub więcej procesów dzieci, kolejność w jakim proces rodzic otrzyma ich status jest niezdefiniowany.

Parametr *stat_loc*, jeśli nie jest ustawiony na *NULL* i jeśli funkcja *wait()* zwróci wartość procesu dziecka to w to miejsce zapiszę się wartość 0 jeśli proces dziecko:

- Zwrócił 0 w *main()*
- Wywołał *exit()* z parametrem 0
- Zakończył się bo wszystkie wątki się zakończyły

Pomimo to wartość ta może być interpretowana makrami mówiące nam co się stało z procesem dzieckiem. Jeśli wartość danego makra jest niezerowa to:

- WIFEXITED – proces zakończył się standardowo

- WEXITSTATUS – Jeśli WIFEXITED niezerowe to można odczytać status z jakim się zakończył
- WIFSIGNALED – Jeśli proces zakończył się z powodu nieprzechwyconego sygnału
- WTERMSIG – Jeśli WIFSIGNALED niezerowe to możemy odczytać numer sygnału
- WIFSTOPPED – Jeśli proces jest aktualnie zatrzymany
- WSTOPSIG – Jeśli WIFSTOPPED niezerowe to możemy odczytać numer sygnału stopu
- WIFCONTINUED – Jeśli status został zwrócony dla procesu ktore kontynuowało z *job control stop*

Jeśli proces rodzic zakończy bez czekania, dzieci otrzymają nowego rodzica (*init*).

Sukces: zwraca PID dziecka. Dostarczono sygnał przerywający: zwraca -1 i errno.

pid_t waitpid(pid_t pid, int *stat_loc, int options) – działa jak *wait()* z kilkoma różnicami.

Czeka na konkretny proces lub grupę procesów, to na jaki proces czeka opisuje **pid** z zasadami takimi jak w funkcji *kill()*.

options to bitowa suma flag, która mówi nam w jaki sposób ma zachować się funkcja. Flagi:

- WCONTINUED – funkcja powinna obsłużyć status dowolnego kontynuowanego procesu, który nie był raportowany odkąd kontynuował po "job control stop".
- WNOHANG – Funkcja nie blokuje wątku i nie czeka, jeśli od razu nie ma do odebrania statusu jakiegoś procesu.
- WUNTRACED – Status dowolnego procesu (zdefiniowanego przez *pid* oczywiście) który został zatrzymany i którego status nie został odebrany odkąd się zatrzymał również powinien zostać odebrany.

Sukces: zwraca PID dziecka. Flaga WNOHANG ustawiona i brak (chwilowy) dzieci do odebrania: zwraca 0. Błąd lub dostarczono sygnał przerywający: zwraca -1 i errno.

unsigned sleep(unsigned seconds) – każe wątkowi czekać daną liczbę sekund, w przypadku przerwania przerwania sygnałem zwraca liczbę "niedospanych sekund", w przeciwnym wypadku 0.

void memset(void *s, int c, size_t n) – kopiuje c do pierwszych n bajtów obiektu wskazanego przez s. Standardowo czyścimy nią pamięć np. struktury sigaction: *memset(&obiekt, 0, sizeof(struct sigaction))*.

unsigned alarm(unsigned seconds) – funkcja generuje SIGALARM do procesu po upłynięciu danej liczby sekund. Jeśli uruchomiona w trakcie działania innego alarm() zwraca pozostałą liczbę sekund do wygenerowania sygnału. Jeśli jest jedynym wywołaniem zwraca 0.

int pthread_create(pthread_t *restric thread, constr pthread_attr_t *restrict attr, void*(*start_routine)(void*), void *restrict arg) – funkcja generalnie tworzy nowy wątek, ma dość skomplikowaną budowę argumentów, więc należy ją opisać po kolei:

- thread – wskaźnik do struktury, gdzie zapisze się TID (thread ID)
- attr – wskaźnik do struktury z atrubutami dla wątku, strukturę trzeba najpierw zaiinicjować (do przeczytania man 3p pthread_attr_init()), a potem warto ją zniszczyć. NULL oznacza domyślne ustawienia. Na początku będziemy używać jedynie podstawowych opcji.

  Najczęstsza funkcja ustawiania – pthread_attr_setdetachstate(pthread_attr_t *atr, int detachstate) , ustawia nam stan odłączenia nowo tworzonych wątków. Gdzie detachstate to makro **PTHREAD_CREATE_JOINABLE** (będziemy czekać na wątek) i **PTHREAD_CREATE_DETACHED** (nie będziemy czekać).
- start_routine – funkcja wykonywana przez wątek, musi mieć sygnature: przyjmuje wskaźnik na void, zwraca wskaźnik na void.
- arg – argument przekazywany do funkcji wątku.

Jeśli sukces: zwraca 0. Porażka: "error number".

int pthread_join(pthread_t thread, void **value_ptr) – funkcja zawiesza działanie wątku, dopóki wątek dany w *thread* się nie zakończy, chyba że już to zrobił.

int pthread_detach(pthread_t thread) – odłącza dany wątek i nie mamy już prawa wykonywać joina.

int mkfifo(const char *path, mode_t mode) – funkcja tworzy nowy plik FIFO w podanej ścieżce. Tryb dostępu jest określany przez *mode*. Jeśli ścieżka prowadzi do symlinka lub FIFO istnieje funkcja się nie powodzi, a errno ma wartość **EEXIST**. Jeśli sukces: zwraca 0. Porażka: -1, FIFO nie jest tworzone i errno ustawione.

int pipe(int fildes[2]) – funkcja tworzy łącze nienazwane i zapisuje deskryptor do czytania w fildes[0] i deskryptor do pisania w fildes[1]. Jeśli sukces: zwraca 0. Porażka -1, pipe nie jest tworzony i ustawia errno.

mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr) – funkcja tworzy połączenie pomiędzy procesem i kolejką komunikatów. Funkcja posiada parametry:

- **name** – wskazanie na napis (c-string) będący nazwą kolejki komunikatów. Ma ona specyficzny format nazwy ścieżkowej. Bardzo ważne żeby zaczynała się od / i nie miała tego znaku nigdzie indziej. Niestosowanie tej reguły jest niezdefiniowane. Poprawną nazwą jest np. */myqueue*.
- **oflag** – tryb tworzenia kolejki tak jak dla plików: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NONBLOCK.
- **mode** – prawa dostępu do kolejki: r i w.
- **attr** – wskazanie do struktury atrybutów kolejki.

Jeśli sukces: zwraca deskryptor kolejki. Porażka: (mqd_t) -1 i ustawia errno.

int mq_close(mqd_t mqdes) – usuwa połączenie powiędzy deskryptorem kolejki, a kolejką. Jeśli istniała subskrypcja na powiadomienie z tą kolejką i procesem wywołującym to jest ona usuwana. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int mq_unlink(const char *name) – usuwa kolejke komunikatów o podanej nazwie nawet jeśli jakieś procesy jeszcze mają ją otwartą. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio) – funkcja dodaje komunikat do kolejki komunikatów. Jeśli kolejka jest pełna, a flaga NON_BLOCK nieustawiona to funkcja się zablokuje, az nie znajdzie się miejsce lub nie przerwie jej sygnał. Jeśli więcej niż jeden wątek czeka na wysłanie i system wspiera Priority Scheduling to pierwszy jest ten, który najdłużej czekał. W trybie nieblokującym zwróci błąd.

- **mqdes** – deskryptor kolejki
- **msg_ptr** – komunikat
- **msg_len** – długość komunikatu
- **msg_prio** – priorytet

Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio, const struct timespec *abstime) – Działa dokładnie tak jak *mq_send()*, ale dodatkowo podajemy *abstime* – czyli moment, w którym zablokowana funkcja zwróci błąd **ETIMEDOUT**, jeśli nie uda się do tego czasu zapisać komunikatu. Bardzo ważne: to nie jest czas po którym funkcja wyjdzie tylko timestamp, w którym to się stanie. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio) – Odbiera najstarszą spośród wiadomości o najwyższym priorytecie z kolejki z deskryptorem *mqdes*. Mechanizm blokowania działa tak samo jak przy zapisywaniu komunikatu.

- **mqdes** – deskryptor kolejki
- **msg_ptr** – wskaźnik do miejsca gdzie komunikat zostanie skopiowany
- **msg_len** – długość odbieranego komunikatu, jeśli mniejsza niż *mq_msgsize* to funkcja zwróci błąd.
- **msg_prio** – jeśli nie jest NULL, priorytet wiadomości zostanie zapisany do miejsca wskazanego przez wskaźnik

Jeśli sukces: zwraca długość odebranej wiadomości i usuwa komunikat z kolejki. Porażka: -1 i ustawia errno.

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr, size_t msg_len, unsigned *restrict msg_prio, const struct timespec *restrict abstime) – funkcja analogiczna do *mq_receive()* w wersji timed (patrz *mq_send()* i *mq_timedsend()*).

int mq_getattr(mqd_t mqdes, struct mq_attr *attr) – funkcja pobiera atrybuty kolejki komunikatów o deskryptorze *mqdes* i zapisuje w miejsce wskaźnika *attr*. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict newattr, struct mq_attr *restrict oldattr) – funkcja ustawia atrybuty kolejki komunikatów o deskryptorze *mqdes* z miejsca wskazanego przez *newattr* wstawiając starą strukturę do miejsca wskazanego przez *oldattr* UWAGA: nowa struktura może zmienić tylko pole *mq_flags* czyli tryb blokowania, zmiana któregokolwiek innego pola struktury zostanie zigorowana. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int mq_notify(mqd_t mqdes, const struct sigevent *notification) – funkcja służy do ustawienia powiadamiania procesu w momencie gdy kolejka komunikatów była pusta i coś się w niej znajdzie (nie za każdą wiadomością!). Jeśli **notification** jest NULL, wtedy anulujemy subskrypcje i inny proces może ją zrobić. Opcja powiadamiania: sygnał, wątek lub nic. Więcej o strukturze sigevent w *man 7 sigevent*. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

struct sigevent – Struktura służąca do powiadamiania dla mechanizmów asynchronicznych, szczegóły w man 7 sigevent. Jej pola to:

- **int sigev_notify** – Metoda notyfikacji: SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD.
- **int sigev_signo** – Sygnał jakim ma być powiadamiany wątek w przypadku SIGEV_SIGNAL.
- **union sigval sigev_value** – Dane przekazywane do wątku w postaci unii o polach: **int sival_int** i **void *sival_ptr**.
- **void (*sigev_notify_function) (union sigval)** – Funkcja używana przy powiadamianiu wątkiem (SIGEV_THREAD).
- **void *sigev_notify_attributes** – Atrybuty dla funkcji przy powiadamianiu wątkiem (SIGEV_THREAD).

W przypadku metody powiadamiania sygnałem, SIGEV_SIGNAL, jeśli sygnał został przechwycony przez handler zarejestrowany z flagą SA_SIGINFO (man 3p sigaction), wtedy jako drugi argument dostaje on strukturę **siginfo_t** z następującymi wartościami na konkretnych polach:

- *si_code* – pole zależne od API dostarczającego powiadomienie
- *si_signo* – pole ustawione na numer sygnału
- *si_value* – zawiera to co wskazane w polu *sigev_value*

int shm_open(...)

void *mmap(...)

int ftruncate(...)

int munmap(...)

int socket(int domain, int type, int protocol) – funkcja powinna uytworzyć niepowiązany socket (gniazdo) związane z konkretną rodziną protokołów i zwrócić deskryptor pliku, który może być używany do wywoływania funkcji na gnieździe.

- *domain* – argument określa "communication domain" czyli określenie z jakiej rodziny protokołów będziemy korzystać. Dla Unixa najczęściej będziemy używać: **AF_UNIX**(lub synonim **AF_LOCAL**) – komunikacja lokalna wewnątrz systemu, lub **AF_INET** – określa rodzinę protokołów IPv4. Można również spotkać wersję z prefixami **PF_**. Każda rodzina protokołów jest powiązana z jedną rodziną adresowania, dlatego istnieją powiązania symboliczne i nie ma znaczenia czy użyjemy wersji AF_... czy PF_... . np. AF_INET PF_INET.
- *type* – typ gniazda, do wyboru: **SOCK_STREAM**, **SOCK_DGRAM** i **SOCK_SEQPACKET**. Przykładowo SOCK_DGRAM udostępnia datagramy używane przez protokół UDP.
- *protocol* – jeśli zostawimy 0, to zostanie przypisany domyślny protokół dla tej rodziny adresów.

Jeśli sukces: zwraca deskryptor pliku. Porażka: -1 i ustawia errno.

int bind(int socket, const struct sockaddr *address, socklen_t address_len) – funkcja przypisuje lokalny adres gniazda, dla gniazda wskazanego przez deskryptor, które aktualnie nie ma żadnego powiązania z żadnym adresem.

- *socket* – deskryptor pliku powiązany z gniazdem.
- *address* – wskaźnik na strukturę **sockaddr** zawierającą adres do którego ma być przypisane gniazdo. To jest dość skomplikowane i dziwne, bo te struktury zwykle nie są typu *sockaddr*, a jedynie pokrywają te strukture bazową, więc trzeba je rzutować. Opis bazowej struktury i najczęściej używanych poniżej.
- *address_len* - określa długość struktury wskazywanej przez *address*.

Jeśli numer portu odczytany z struktury *address* to system wybiera automatycznie **port efemeryczny**, czyli po prostu automatycznie przydziela mu numer portu.

Jeżeli rodzina adresowania to **AF_UNIX** i ścieżka do pliku w adresie jest symlinkiem to funkcja się nie powiedzie.

Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

struct sockaddr – ogólna struyktura adresowa, musi zawierać pola:

- *sa_family_t* ***sa_family*** – rodzina adresowa. Używamy makr np. AF_UNIX.
- *char* ***sa_data_t[]*** – adres właściwy dla protokołu, zmienna długość.

struct sockaddr_in – struktura adresowa dla IPv4, musi zawierać pola:

- *sa_family_t* ***sin_family*** – koniecznie == AF_INET.
- *in_port_t* ***sin_port*** – 16 bajtowy numer portu (uint16_t)
- *struct in_addr* ***sin_addr*** – struktura zawierająca tylko jedno pole: 32 bajtowy adres IPv4 (uint32_t)

Ważne: sin_port i sin_addr są w porządku sieciowym (big-endian).

struct sockaddr_un – struktura adresowa dla domeny UNIX, musi zawierać pola:

- *sa_family_t* ***sun_family*** – koniecznie == AF_UNIX lub AF_LOCAL.
- *char* ***sun_path[]*** – nazwa ścieżkowa, w POSIX długość nieokreślona, zwykle 92-108. Jest to ścieżka bewzględna!

int connect(int socket, const struct sockaddr *address, socklen_t address_len) – funkcja próbuje nawiązać połączenie na gnieździe w trybie połączeniowym lub w przypadku gniazd w trybie bezpołączeniowym ustawia lub resetuje adres "partnera". (peer).

- *socket* – deskryptor pliku powiązany z gniazdem.
- *address* – wskaźnik na strukturę **sockaddr** zawierającą adres hosta z którym chcemy nawiązać połączenie.
- *address_len* - określa długość struktury wskazywanej przez *address*.

Jeśli gniazdo nie jest wcześniej zaadresowane to funkcja automatycznie go zaadresuje, (nie działa rodzinie adresowej AF_UNIX). Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int listen(int socket, int backlog) – funkcja oznacza gniazdo o deskryptorze *socket* w trybie połączeniowym jako akceptujące połączenia – rozpoczyna nasłuch. Parametr *backlog* określa limit oczekujących połączeń w kolejce oczekujących dla gniazda. Jeśli sukces: zwraca 0. Porażka: -1 i ustawia errno.

int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len) – funkcja wybiera pierwsze połączenie w kolejce oczekujących, tworzy nowe gniazdo o takim samym typie protokołu i rodzinie adresowej jak wybrane gniazdo i alokuje nowy deskryptor pliku dla tego gniazda.

- *socket* – określa deskryptor gniazda, które słuchało funkcją *listen()* i właśnie otrzymało połącznenie.
- *address* – wskaźnik na strukturę **sockaddr**, gdzie zapisze się adres podłączającego się socketa. Może być NULL.
- *address_len* - wskaźnik na **socklen_t** gdzie zapisze się długość struktury adresowej. Może być NULL.

Jeśli sukces: zwraca deskryptor pliku powiązany z zaakceptowanym gniazdem. Porażka: -1 i ustawia errno.

int pselect(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict errorfds, const struct timespec *restrict timeout, const sigset_t *restrict sigmask) – funkcja powinna deskryptory podane w formie zestawów w polach: readfds, writefds i errordfs aby sprawdzić czy któreś z nich są gotowe do odczytu, do zapisu, albo mają oczekujące wyjątki (dokładnie w tej kolejności – każdy zestaw badamy inaczej).

nfds określa zakres deskryptorów plików do sprawdzenia (sprawdza się od 0 do nfds-1 czy jakoś tak). W przypadku powodzenia, funkcja modyfikuje obiekty wskazywane przez wskaźniki w parametrach pokazując, który deskryptor jest gotowy do czytanie, pisania, obsługi błędów.

Funkcja zastąpi aktualną maskę, maskami podanymi w parametrze *sigmask* (jeśli nie NULL). Można ustawić również max timeout, albo zostawić NULL.

Funkcja zwróci całkowitą liczbe deskryptorów. Jeśli żaden deskryptor nie jest gotowy to funkcja się zablokuje.

int fcntl(int fildes, int cmd, ...) – funkcja wykonuje operacje na otwartych plikach. *fildes* to deskryptor plikum, a *cmd* to flagi. Przykładowo: F_GETFL zwróci nam aktualne flagi pliku, a F_SETFL ustawi nowe flagi. Funkckja zwraca różne wartości, w zależności od podanych flag *cmd*.

void freeaddrinfo(struct addrinfo *ai) – funkcja zwalnia struktury addrinfo zwrcane przez funkcje getaddrinfo().

int getaddrinfo(const char *restrict nodename, const char *restrict servname, const struct addrinfo *restrict hints, struct addrinfo **restrict res) – funkcja tłumaczy nazwy serwisów na adresy gniazd. Dość obszerne narzędzie do np. znajdywania adresów ip.

- *nodename* – jeśli nie jest nullem, nazwa lub adres, np : google.com
- *servname* – jeśli nie jest nullem, to mamy tutaj pożądaną usługę np. dla AF_INET będzie to numer portu.
- *hints* - jeśli nie jest nullem, odnosi się do struktury z ustawieniami.
- *res* - wskaźnik do struktury z wypełnionymi polami określającymi adres gniazda i informacje potrzebne do utworzenia gniazda.

Zwraca błąd, który możemy odczytać funkcją gar_strerror

const char *gai_strerror(int ecode) – Pozwala otrzymać wiadomość błędu opisujący błąd z funkcji getaddrinfo().