Systemy Operacyjne 2024/2025

Projekt „Rejs”

Wiktoria Herczyk 151431

SPIS TREŚCI

O PROJEKCIE

Co zawiera projekt:

Na projekt składają się:

- 3 programy (kapitan.c, pasażer.c, kapitan_portu.c)
- Biblioteka z zainicjowanymi funkcjami (rejs.h)
- Program z definicjami funkcji z rejs.h (rejs.c)
- Plik Makefile

Opis projektu:

Przy nabrzeżu stoi statek o pojemności N pasażerów. Statek z lądem jest połączony mostkiem o pojemności K (K<N). Na statek próbują dostać się pasażerowie, z tym, że na statek nie może ich wejść więcej niż N, a wchodząc na statek na mostku nie może być ich równocześnie więcej niż K. Statek co określoną ilość czasu T1 (np.: jedną godzinę) wypływa w rejs. W momencie odpływania kapitan statku musi dopilnować aby na mostku nie było żadnego wchodzącego pasażera. Jednocześnie musi dopilnować by liczba pasażerów na statku nie przekroczyła N. Dodatkowo statek może odpłynąć przed czasem T1 w momencie otrzymania polecenia (sygnał1) od kapitana portu. Rejs trwa określoną ilość czasu równą T2. Po dotarciu do portu pasażerowie opuszczają statek. Po opuszczeniu statku przez ostatniego pasażera, kolejni pasażerowie próbują dostać się na pokład (mostek jest na tyle wąski, że w danym momencie ruch może odbywać się tylko w jedną stronę). Statek może wykonać maksymalnie R rejsów w danym dniu lub przerwać ich wykonywanie po otrzymaniu polecenia (sygnał2) od kapitana portu (jeżeli to polecenie nastąpi podczas załadunku, statek nie wypływa w rejs, a pasażerowie opuszczają statek. Jeżeli polecenie dotrze do kapitana w trakcie rejsu statek kończy bieżący rejs normalnie).

Instrukcja obsługi:

Program był pisany i testowany na Ubuntu 20.04 (później również na Torusie).

Instrukcja Makefile:

- „make” – kompiluje programy
- „make testx” – uruchamia symulacje z danymi do odpowiedniego testu
- „make user” – uruchamia symulacje z danymi do wprowadzenia przez użytkownika
- „make clean” – czyszczenie

Program powinno się uruchamiać w kolejności:

- Pasażer -> Kapitan -> Kapitan Portu

Dane przyjmowane przez programy:

- Kapitan:
  - pojemność mostka
  - pojemność statku
  - ilość rejsów
  - czas jednego rejsu

- Kapitan portu:
    - czas między rejsami
    - czas jednego rejsu
    - szansa na burzę

DZIAŁANIE PROGRAMÓW

Moja interpretacja i implementacja mechaniki sygnałów:

Sygnał1 – sygnał start – jest wysyłany zarówno regularnie po upłynięciu czasu T1 jak i w przypadku, gdy Kapitan Portu zezwoli na wcześniejsze odpłynięcie. (By nie było to zupełnie zdarzenie losowe, założyłam że warunkiem do rozważenia wcześniejszego startu jest zapełnienie się całego statku przed upływem T1)

Sygnał2 – sygnał stop – jest wysyłany losowo, w zależności od podanych danych wejściowych tj. szansy na burzę w danej symulacji i czasu T1. (Dodałam funkcjonalność która losowo może wygenerować burzę/sztorm na morzu – według procentowego prawdopodobieństwa podanego przez użytkownika. Gdy burza wystąpi, rejsy się kończą. Tak naprawdę jest to głównie element kosmetyczny, by sygnał nie był wysyłany kompletnie losowo i teoretycznie bezpodstawnie).

Oprócz tego, każdy proces reaguje indywidualnie na sygnał SIGINT w sposób, który kończy jego działanie i w miarę możliwości po nim ‘sprząta’.

Zadania poszczególnych procesów:

- Pasażer:
    - Tworzy zbiór semaforów
    - Dołącza do kolejki komunikatów
    - Co losowy czas (1-5s) tworzy podprocesy (faktycznych Pasażerów)
        - Wchodzi na mostek (przesyła swój PID do kolejki komunikatów)
        - Schodzi z mostka
- Kapitan:
    - Przekazuje swój PID Kapitanowi portu
    - Odbiera PID Kapitana portu
    - Ustawia mostek (tworzy kolejkę komunikatów)
    - Tworzy zbiór semaforów
    - Wpuszcza pasażerów na statek (odbiera PID pasażera z kolejki i przechowuje go w tablicy)
    - Pilnuje, by mostek był pusty przed startem
    - Wypływa w rejs (sleep)
    - Wypuszcza pasażerów ze statku (przesyła PID pasażera do kolejki po zakończonym rejsie)
    - Pilnuje, by po skończonym rejsie Pasażerowie najpierw zeszli, zanim zaczną wchodzić kolejni
    - Nasłuchuje komunikatów Kapitana portu (odbiera sygnały do startu i przerwania rejsów i zamyka wejście na mostek)

- Kapitan portu:
  - Odbiera PID Kapitana
  - Przekazuje swój PID Kapitanowi statku
  - Działa równocześnie na dwóch wątkach:
    1) - Odlicza czas do startu i wysyła sygnały do rozpoczęcia rejsu
       - Jeśli statek zapełni się w trakcie ustalonej przerwy między rejsami kapitan wyśle sygnał do startu przed czasem
    2) - Wypatruje sztormu (dwa razy na czas trwania rejsu losuje czy zakończyć kursowanie statku wcześniej)

Dane wprowadzane przez użytkownika i ich limity:

T1 – czas (między rejsami) – czas jaki statek czeka w porcie (może być skrócony po sygnale Kapitana Portu);

T2 – czas_rejsu – czas trwania rejsu, krótszy od czasu T1;

N – pojemność_statku – maksymalna ilość pasażerów, która może na raz znajdować się na statku w tym samym czasie i brać udział w pojedynczym rejsie;

K – pojemność_mostka - maksymalna ilość pasażerów, która może na raz znajdować się na mostku w tym samym czasie;

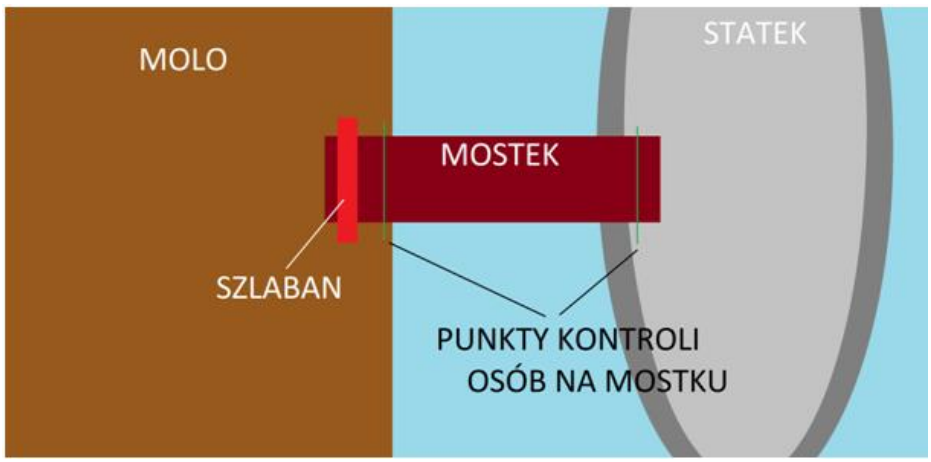
R – ilość_rejsów_dziś – ile maksymalnie rejsów może się odbyć podczas symulacji;

szansa_na_burze – procentowa szansa, że Kapitan Portu podczas jednego rejsu zauważy sztorm i przerwie kursowanie statku; [0, 100]

Poza szansą na burzę, wszystkie wartości wprowadzane przez użytkownika muszą być większe od zera.

Użyte struktury i ich wizualizacja:

- Tablica PIDów – statek
- Kolejka komunikatów – mostek na statek
- Semafory – szlaban do wejścia na mostek i punkty kontroli osób (SZLABAN (1) – wielkości miejsc na statku i MOST (0))
- Potok FIFO – sposób do przekazania PIDów między kapitanami



Wyjaśnienie logiki programu:

Semafor SZLABAN kontroluje ile osób może wejść na dany rejs. Domyślnie jego wartość wynosi tyle co maksymalna ilość pasażerów jednego rejsu. Każdy pasażer chcąc dostać się na mostek obniża wartość tego semafora. W ten sposób nikt nie wpycha się na siłę na mostek, gdy nie ma dla niego miejsca na statku. SZLABAN osiąga 0, gdy cały statek się zapełni lub gdy kapitan otrzyma komunikat do startu, a na statku wtedy będzie przynajmniej jeden pasażer.

Zaraz po przejściu 'za szlaban' proces zwiększa wartość semafora MOST. Dzięki temu kapitan wie, ile osób jest zakwalifikowanych na rejs – i ile osób ma wejść na statek przed jego rozpoczęciem. W ten sposób nigdy nie trzeba nikogo z mostka wypraszać, ani nikt na nim nie zostanie przed odpłynięciem. Po wejściu pasażera na statek kapitan obniża wartość MOST.

Dokumentacja funkcji zawartych w rejs.c i rejs.h

Funkcja	int utworz_semafor(key_t klucz, int nr)
Co zwraca	Identyfikator semafora (ID) w przypadku sukcesu (int)
Gdy błąd	Program kończy działanie

Tworzy semafor z podanym kluczem i liczbą semaforów

Parametry:

key_t klucz: Klucz IPC służący do identyfikacji zestawu semaforów.

int nr: Liczba semaforów w zestawie.

Funkcja	void ustaw_wartosc_semafora(int wartosc, int nr, int sem)
Co zwraca	Nic (funkcja typu void)
Gdy błąd	Program kończy działanie

Ustawia wartość określonego semafora w zestawie.

Parametry:

int wartosc: Wartość, którą należy przypisać semaforowi.

int nr: Indeks semafora w zestawie.

int sem: Identyfikator zestawu semaforów.

Funkcja	int sprawdz_wartosc_semafora(int nr, int s)
Co zwraca	Wartość semafora (int)
Gdy błąd	-1

Pobiera wartość określonego semafora.

Parametry:

int nr: Indeks semafora w zestawie.

int s: Identyfikator zestawu semaforów.

Funkcja	void zakoncz(int kolejka, int semafor, pid_t pid)
Co zwraca	Nic (funkcja typu void)

Usuwa kolejkę wiadomości, semafor oraz plik FIFO, a następnie wysyła sygnał do procesu.

Parametry:

int kolejka: Identyfikator kolejki wiadomości.

int semafor: Identyfikator zestawu semaforów.

pid_t pid: PID procesu, do którego zostanie wysłany sygnał.

Funkcja	void wyslij_pid(pid_t pid, const char *fifo_path)
Co zwraca	Nic (funkcja typu void)
Gdy błąd	Program kończy działanie

Wysyła PID procesu do pliku FIFO.

Parametry:

pid_t pid: PID procesu do wysłania.

const char *fifo_path: Ścieżka do pliku FIFO.

Funkcja	pid_t odbierz_pid(const char *fifo_path);
Co zwraca	PID odczytany z pliku FIFO (pid_t)
Gdy błąd	Program kończy działanie

Odczytuje PID procesu z pliku FIFO.

Parametry:

const char *fifo_path: Ścieżka do pliku FIFO.

Funkcja	void wyslij_sygnal(pid_t pid, int sygnal)
Co zwraca	Nic (funkcja typu void)
Gdy błąd	Program kończy działanie

Wysyła sygnał do określonego procesu.

Parametry:

pid_t pid: PID procesu, do którego należy wysłać sygnał.

int sygnal: Numer/Kod sygnału do wysłania.

Funkcja	int polacz_kolejke(int s)
Co zwraca	Identyfikator kolejki wiadomości. (int)
Gdy błąd	-1

Łączy się z istniejącą kolejką wiadomości.

Parametry:

int s: Identyfikator semafora używanego w przypadku błędu.

Funkcja	void otworz_fifo(const char *fifo_path, int *fd, int mode)
Co zwraca	Nic (funkcja typu void)
Gdy błąd	Program kończy działanie

Otwiera plik FIFO w określonym trybie.

Parametry:

const char *fifo_path: Ścieżka do pliku FIFO.

int *fd: Wskaźnik na deskryptor pliku.

int mode: Tryb otwarcia.

Opis działania głównych funkcji każdego z programów:

Kapitan.c:

1. Start
   Początek programu.
2. Sprawdzenie liczby argumentów
   Decyzja: Czy liczba argumentów wynosi 4?
   Tak: Przejdź dalej.
   Nie: Wyświetl komunikat o błędzie i zakończ program.
3. Inicjalizacja danych wejściowych
   Pobranie wartości argumentów:
   pojemnosc_mostka
   pojemnosc_statku
   ilosc_rejsow_dzis
   czas_rejsu
4. Walidacja danych wejściowych
   Decyzja: Czy wartości wejściowe są nieujemne?
   Nie: Wyświetl komunikat o błędzie i zakończ program.
   Decyzja: Czy pojemnosc_mostka < pojemnosc_statku?
   Nie: Wyświetl komunikat o błędzie i zakończ program.
5. Przygotowanie komunikacji
   Tworzenie pliku FIFO.
   Wysłanie i odbieranie PID-ów (kapitana portu).
   Usunięcie FIFO.
6. Konfiguracja kolejki i semaforów
   Dołączanie do kolejki komunikatów.
   Tworzenie semaforów.
   Ustawienie limitu rozmiaru kolejki (mostka).
7. Obsługa rejsów
7.1 Pętla główna
   Decyzja: Czy liczba rejsów na dziś > 0 i flaga plyn aktywna?
   Tak: Rozpocznij obsługę rejsu.
   Nie: Przejdź do zakończenia programu.
7.2 Obsługa pasażerów
   Pętla: Pobieranie pasażerów z kolejki:
   Jeśli liczba pasażerów < pojemność statku: Dodaj pasażera na statek.
   Inaczej: Przerwij pętlę.
   Pętla: Upewnij się, że kolejka mostka jest pusta.
7.3 Czekanie na sygnał startu
   Pętla: Czekanie na sygnał startu (jeśli flaga startuj = 0).
7.4 Symulacja rejsu
   Decyzja: Czy liczba pasażerów > 0?
   Tak: Symuluj rejs, zmniejsz liczbę pozostałych rejsów.
   Nie: Wyświetl komunikat, kontynuuj.
7.5 Wyładunek pasażerów
   Pętla: Wyładuj pasażerów ze statku.
7.6 Przygotowanie do następnego rejsu
   Zresetuj liczbę pasażerów.
   Wyzeruj flagę startuj.
8. Zakończenie programu
   Decyzja: Czy wszystkie rejsy wykonano?
   Tak: Wyświetl komunikat o zakończeniu rejsów.
   Nie: Wyświetl komunikat o wstrzymaniu rejsów.
Zakończ program.

Pasażer.c:

Program główny:

1. Start
   Początek programu.
2. Konfiguracja kolejki i semaforów
   Utworzenie kolejki komunikatów.
   Tworzenie semaforów.
3. Obliczenie wartości clear (maksymalna liczba ostatnich procesów potomnych w tablicy pidów)
4. Pętla główna programu (while (1))
   Decyzja: Czy semafory zostały usunięte?
   TAK: Zakończ pętle.

4.1 Tworzenie nowych pasażerów
   Decyzja: ilosc_pasazerow < limit.
   TAK:
   Wygeneruj losowy czas oczekiwania (czas_miedzy_pasazerami).
   Odczekaj (sleep(czas_miedzy_pasazerami)).
   Utwórz proces potomny (fork()).

5. Zakończenie programu
   Oczekuj zakończenia wszystkich procesów potomnych.
   Wyświetl komunikat o zakończeniu procesów.
   Zakończ program.

Program potomny:

1. Obniż wartość semafora SZLABAN.
2. Zwiększ wartość semafora MOST.
3. Wyślij pasażera (pid) do kolejki komunikatów.
4. Obniż wartość semafora MOST.
5. Oczekiwanie na wiadomość zwrotną od kapitana (na zakończenie rejsu)
6. Zakończenie programu

Kapitan_portu.c:

1. Start
   Początek programu.
2. Sprawdzenie liczby argumentów
   Decyzja: Czy liczba argumentów wynosi 3?
   Tak: Przejdź dalej.
   Nie: Wyświetl komunikat o błędzie i zakończ program.
3. Inicjalizacja danych wejściowych
   Pobranie wartości argumentów:
   czas
   czas_rejsu
   szansa na burzę
4. Walidacja danych wejściowych
   Decyzja: Czy wartości wejściowe są nieujemne?
   Nie: Wyświetl komunikat o błędzie i zakończ program.
   Decyzja: Czy czas_rejsu < czas?
   Nie: Wyświetl komunikat o błędzie i zakończ program.
5. Przygotowanie komunikacji
   Tworzenie pliku FIFO.
   Wysłanie i odbieranie PID-ów (kapitan).
   Usunięcie FIFO.
6. Konfiguracja kolejki i semaforów
   Tworzenie semaforów.
7. Tworzenie wątków
   Utwórz wątek sygnal_start dla wyslij_sygnal_start().
   Warunek: Wątek nie został utworzony?
   TAK: Wypisz komunikat o błędzie i zakończ program.
   Utwórz wątek sygnal_stop dla wyslij_sygnal_stop().
   Warunek: Wątek nie został utworzony?
   TAK: Wypisz komunikat o błędzie i zakończ program.
8. Oczekiwanie na zakończenie wątków
9. Zakończenie programu
   Zakończ program.

Wątek do sygnału start:

1. Odczekaj czas rejsu
2. Oblicz różnicę między czasami (pom_czas)
3. Pętla główna
   Decyzja: Czy flaga plyn aktywna?
   Tak: Rozpocznij odliczanie do startu
   Nie: Przejdź do zakończenia wątku

3.1. Obsługa sygnału
   Pętla: oczekiwania na kolejny rejs (for i < pom_czas)
   Opóźnienie: sleep(1).
   Sprawdzenie wartości semafora
   Warunek: Statek jest już cały zapełniony.
   TAK: Wyświetl komunikat i przerwij pętlę for.
   Wyślij sygnał START do kapitana

3.2. Sprawdzenie istnienia semafora
   Decyzja: Czy semafory zostały usunięte?
   TAK: Wyślij sygnał SIGINT i zakończ program.

3.3. Czekanie na zakończenie rejsu
4. Zakończenie programu

Wątek do sygnału stop:

1. Pętla główna
   Decyzja: Czy flaga plyn aktywna?
   Tak: Rozpocznij odliczanie do startu
   Nie: Przejdź do zakończenia wątku

1.1. Obsługa sygnału
   Pętla: oczekiwania na kolejny rejs (for i < czas/2)
   Opóźnienie: sleep(1).
   Losowanie czy jest burza
   Decyzja: Czy wylosowana liczba < szansa na burzę?
   TAK: Wyślij sygnał STOP do kapitana
   Wyświetl komunikat i przerwij pętle

2. Zakończenie programu

TESTY

Testy zawarte w pliku Makefile:

test1:

./p &

./k 5 10 1 20 &

./kp 5 20 0 &

Test sprawdza obsługę wyjątków – podane dane są niepoprawne (czas między rejsami powinien być większy niż czas rejsu – tutaj 5 < 20), program powinien zwrócić o tym informację i zakończyć działanie nie zostawiając żadnych działających procesów czy struktur

test2:

./p &

./k 5 60 5 10 &

./kp 20 10 0 &

Test sprawdzający czy statek odpływa co określony czas, nieważne czy jest w on cały zapełniony i czy symulacja kończy się po odbyciu wszystkich rejsów. Miejsce na statku jest za duże, by zostało zapełnione w czasie pomiędzy rejsami nawet jeśli pasażerowie przychodzili by jeden po drugim (czyli co sekundę). Mechanizm burzy został wyłączony.

test3:

./p &

./k 2 5 5 1 &

./kp 60 1 0 &

Test sprawdzający mechanikę odpływania przed czasem, gdy statek jest pełen. Mimo, że cały czas miedzy rejsami nie minął, statki powinny odpływać po wpuszczeniu 5 pasażerów oraz program powinien wyświetlać komunikat o wcześniejszym starcie. Mechanizm burzy został wyłączony.

test4:

./p &

./k 2 60 5 1 &

./kp 11 1 100 &

Test sprawdzający mechanikę burzy – losowego przerwania programu. Rejsy powinny zostać przerwane, a jeśli statek stoi w porcie to każdy kto na zdążył się na niego dostać powinien z niego zejść.

Opisy reszty wykonanych testów:

Program został sprawdzony w sytuacji, gdy wszystkie sleepy są wyłączone. Dla bezpieczeństwa i lepszej czytelności oprócz sleepów zakomentowałam też informacje o pojawiających się pasażerach i ciągłych sygnałach do startu. Na moim systemie bez problemu dał radę wykonać symulację dla limitu ustawionego na 20000 pasażerów i 1000 rejsów po 5 pasażerów.

Przeprowadziłam prawdziwą symulację (z włączonymi sleep'ami) trwającą około 20 minut.

(Z parametrami: rozmiar mostka: 2;
rozmiar statku: 5;
czas rejsu: 1s;
czas między rejsami: 5s,
ilość rejsów: 300;
szansa na burzę 0%)

Spróbowałam losowo wstrzymywać procesy CTRL+Z. Nie spowodowało to zniszczenia się całej symulacji.

Sprawdziłam jak zachowują się programy, po fizycznym usunięciu. Po usunięciu jednego symulacja się zatrzymuje. Po usunięciu wszystkich w pamięci nic nie powinno zostać.

Największe parametry jakie testowałam na torusie (bez sleepów): ilość pasażerów 20000, ilość rejsów 100).

POZOSTAŁE INFORMACJE

Czego nie udało się zaimplementować:

Próbowałam jeszcze dodać interfejs graficzny. Jednak okazało się to zbyt skomplikowane w języku C, a same próby implementacji (czy to przez połączenie z pytonem czy z użyciem specjalnych bibliotek w C) zabierały za dużo czasu.

Chciałam też, by kapitan miał losową szansę na niezgodzenie się na wcześniejszy start. Jednak okazało się to trudniejsze do zsynchronizowania niż myślałam.

Link do GitHub:

Repozytorium: https://github.com/Tori394/SOprojekt_REJS

Linki do przykładów użycia istotnych funkcji w kodzie:

- fork()
- wait()
- pthread_create()
- pthread_join()
- kill()
- signal()
- semget()

- semctl()
- semop()
- mkfifo()
- msgget()
- msgsnd()
- msgrcv()
- msgctl()