

# Raport - Symulator Tramwaju Wodnego

**Przedmiot:** Systemy Operacyjne

**Projekt:** Tramwaj wodny Kraków Wawel - Tyniec

**Autor:** Szymon Rafałowski

**Repozytorium:** <https://github.com/szyraf/tramwaj-wodny>

## 1. Założenia projektowe

### 1.1. Opis problemu

Symulacja tramwaju wodnego kursującego między przystankami Tyniec i Wawel. Program modeluje:

- Pasażerów czekających w kolejkach na obu przystankach
- Statek o ograniczonej pojemności (N osób, M rowerów)
- Mostek (trap) o ograniczonej przepustowości (K slotów)
- Kapitana zarządzającego fazami rejsu
- Dyspozytora obsługującego sygnały sterujące

### 1.2. Parametry konfiguracyjne

Parametr	Opis
N	Pojemność statku (osoby)
M	Pojemność statku (rowery)
K	Pojemność mostka
T1	Maksymalny czas załadunku (ms)
T2	Czas podróży (ms)
R	Liczba rejsów dziennie

## 1.3. Architektura wieloprocesowa

Program składa się z niezależnych procesów:

1. **Main** - proces główny, tworzy IPC i spawnuje procesy potomne
2. **Captain** - zarządza fazami: załadunek → podróż → rozładunek
3. **Dispatcher** - obsługuje sygnały z klawiatury (1=wcześniejszy odpływ, 2=koniec dnia)
4. **Passenger** (N procesów) - każdy pasażer to osobny proces

## 2. Ogólny opis kodu

### 2.1. Struktura plików

```
src/  
└── main.cpp      - Inicjalizacja IPC, fork() procesów  
└── captain.cpp   - Logika kapitana (fazy rejsu)  
└── passenger.cpp - Automat stanowy pasażera  
└── dispatcher.cpp - Obsługa klawiatury  
└── ipc.cpp/h     - Wrappery System V IPC  
└── config.cpp/h   - Parsowanie konfiguracji + walidacja  
└── logger.cpp/h   - Logowanie z timestampem  
└── common.h       - Struktury danych, enumy
```

### 2.2. Mechanizmy IPC

Program wykorzystuje **trzy mechanizmy IPC System V**:

1. **Pamięć współdzielona** - przechowuje stan symulacji ( SharedState )
2. **Semafony** - synchronizacja (mutex + semafor per pasażer)
3. **Kolejki komunikatów** - potwierdzenia akcji (ACK)

### 2.3. Fazy symulacji

PHASE\_LOADING → PHASE\_BRIDGE\_CLEAR → PHASE\_SAILING → PHASE\_UNLOADING  
↑ |  
+----- (następny rejs) -----+

## 2.4. Stany pasażera

STATE\_QUEUE → STATE\_BRIDGE → STATE\_SHIP → STATE\_BRIDGE → STATE\_EXITED

## 3. Co udało się zrobić

- Pełna symulacja z wieloma procesami (fork + exec)
- Synchronizacja przez semafory System V
- Komunikacja przez pamięć współdzieloną + kolejki komunikatów
- Obsługa dwóch sygnałów (Signal1, Signal2)
- Walidacja danych wejściowych (limity procesów, parametry)
- Obsługa błędów z perror()
- Minimalne prawa dostępu (0600)
- Sprzątanie zasobów IPC po zakończeniu
- Logowanie z timestampem do pliku i terminala
- Obsługa pasażerów z rowerami (zajmuje 2 sloty na mostku)

## 4. Kluczowe decyzje projektowe

### 4.1. Ochrona sekcji krytycznych

Dostęp do SharedState wymaga synchronizacji - zastosowano semafor SEM\_MUTEX jako mutex, aby tylko jeden proces mógł modyfikować współdzielone dane w danym momencie.

### 4.2. Mechanizm oprozniania mostka

Przed odpłynięciem statku pasażerowie pozostający na mostku muszą wrócić do kolejki.

Wprowadzono fazę PHASE\_BRIDGE\_CLEAR, w której kapitan aktywnie budzi pasażerów semaforami.

### 4.3. Obsługa przerwan systemowych (EINTR)

Sygnały (np. SIGCHLD) mogą przerwać wywołania semop() i msgrecv(). Operacje IPC są powtarzane w pętli while errno == EINTR.

## 4.4. Sprzatanie zasobow IPC

Zasoby System V IPC pozostają w systemie po zakonczeniu procesu. Program obsługuje `SIGINT / SIGTERM` i wywoluje `cleanup_ipc()` również na starcie (usuwa pozostałości po poprzednim uruchomieniu).

## 5. Elementy specjalne

### 5.1. Dynamiczna liczba semaforów

Każdy pasażer ma dedykowany semafor do budzenia. Liczba semaforów =  $2 + \text{liczba\_pasażerów}$ .

### 5.2. Blokada pliku logów ( `flock` )

Wieloprocesowy zapis do pliku logu z synchronizacją przez `flock()` .

### 5.3. Sprawdzanie limitu procesów

Przed startem program sprawdza `RLIMIT_NPROC` , aby nie przekroczyć limitu systemowego.

### 5.4. Timestampy względne

Logi pokazują czas od startu symulacji (nie czas systemowy).

## 6. Testy

### Test 1: Podstawowy ( `basic.env` )

**Cel:** Prosta symulacja z kilkoma pasażerami

**Konfiguracja:** N=5, M=2, K=3, T1=5s, T2=1s, R=2, 6 pasażerów

**Wynik:** PASS - wszyscy pasażerowie przewiezieni, 2 rejsy wykonane

## **Test 2: Pełny statek ( full\_ship.env )**

**Cel:** Sprawdzenie limitu pojemności statku

**Konfiguracja:** N=3, K=2, R=3, 6 osób w Tyńcu

**Wynik:** PASS - statek nigdy nie przekracza 3 osób, osoby na mostku wracają do kolejki

## **Test 3: Signal1 - wcześniejszy odpływ ( signal1.env )**

**Cel:** Wczesne odpłynięcie po naciśnięciu '1'

**Konfiguracja:** N=10, T1=30s, dużo pasażerów

**Wynik:** PASS - statek odpływa natychmiast po Signal1

## **Test 4: Signal2 - koniec dnia ( signal2.env )**

**Cel:** Zakończenie dnia po naciśnięciu '2'

**Konfiguracja:** N=10, R=10, dużo pasażerów

**Wynik:** PASS - symulacja kończy się przed wykonaniem wszystkich rejsów

## **Test 5: Rowery ( bikes.env )**

**Cel:** Sprawdzenie limitu rowerów (M)

**Konfiguracja:** N=5, M=2, K=4, 2 osoby + 4 z rowerami

**Wynik:** PASS - max 2 rowery na statku

## **Test 6: Obciążeniowy ( stress.env )**

**Cel:** Stabilność przy dużej liczbie pasażerów

**Konfiguracja:** N=50, M=20, K=30, R=20, 300 pasażerów

**Wynik:** PASS - brak zawieszenia, program kończy się normalnie

## **7. Linki do kodu (GitHub)**

### **7.1. Tworzenie i obsługa plików**

Funkcja	Lokalizacja
fopen()	logger.cpp:48

Funkcja	Lokalizacja
fclose()	logger.cpp:54
read()	dispatcher.cpp:37
flock()	logger.cpp:50-53

## 7.2. Tworzenie procesów

Funkcja	Lokalizacja
fork()	main.cpp:119, main.cpp:128, main.cpp:138
execl()	main.cpp:122, main.cpp:131, main.cpp:143
exit() / _exit()	main.cpp:34, main.cpp:124
wait()	main.cpp:156
waitpid()	main.cpp:24

## 7.3. Obsługa sygnałów

Funkcja	Lokalizacja
signal()	main.cpp:111-113
kill()	main.cpp:30
Handler SIGINT/SIGTERM	main.cpp:27-35
Handler SIGCHLD	main.cpp:22-25

## 7.4. Synchronizacja procesów (semafony)

Funkcja	Lokalizacja
semget()	ipc.cpp:45, ipc.cpp:54
semctl()	ipc.cpp:90, ipc.cpp:97, ipc.cpp:106
semop()	ipc.cpp:64, ipc.cpp:73

Funkcja	Lokalizacja
Użycie mutex	<a href="#">captain.cpp:61</a> , <a href="#">passenger.cpp:110</a>

## 7.5. Segmenty pamięci współdzielonej

Funkcja	Lokalizacja
shmget()	<a href="#">ipc.cpp:6</a> , <a href="#">ipc.cpp:15</a>
shmat()	<a href="#">ipc.cpp:24</a>
shmdt()	<a href="#">ipc.cpp:33</a>
shmctl()	<a href="#">ipc.cpp:39</a> , <a href="#">main.cpp:13</a>

## 7.6. Kolejki komunikatów

Funkcja	Lokalizacja
msgget()	<a href="#">ipc.cpp:112</a> , <a href="#">ipc.cpp:121</a>
msgsnd()	<a href="#">ipc.cpp:133</a>
msgrcv()	<a href="#">ipc.cpp:143</a>
msgctl()	<a href="#">ipc.cpp:154</a> , <a href="#">main.cpp:19</a>

## 7.7. Obsługa błędów i walidacja

Element	Lokalizacja
perror()	<a href="#">ipc.cpp:8</a> , <a href="#">ipc.cpp:17</a> , <a href="#">main.cpp:120</a>
Walidacja konfiguracji	<a href="#">config.cpp:64-102</a>
Sprawdzanie RLIMIT_NPROC	<a href="#">config.cpp:65-82</a>
Obsługa EINTR	<a href="#">ipc.cpp:64-68</a>

## 7.8. Podział na moduły

Moduł	Odpowiedzialność
main.cpp	Inicjalizacja, tworzenie procesów
captain.cpp	Logika kapitana
passenger.cpp	Automat stanowy pasażera
dispatcher.cpp	Obsługa klawiatury
ipc.cpp	Wrappery IPC
config.cpp	Konfiguracja
logger.cpp	Logowanie

## 8. Podsumowanie użytych konstrukcji

Wymaganie	Status	Opis
Tworzenie procesow	TAK	fork() , exec1() , wait() , waitpid()
Mechanizmy synchronizacji	TAK	Semafory System V ( semget , semop , semctl )
Dwa mechanizmy IPC	TAK	Pamiec wspoldzielona + kolejki komunikatow
Obsluga sygnalow	TAK	SIGINT, SIGTERM, SIGCHLD + Signal1/Signal2
Walidacja danych	TAK	validate_config() w config.cpp
Podzial na moduły	TAK	7 plikow zrodlowych
Obsluga bledow	TAK	perror() , errno , obsluga EINTR
Minimalne prawa dostepu	TAK	0600 dla wszystkich zasobow IPC
Sprzatanie zasobow	TAK	cleanup_ipc() + obsluga sygnalow

## 9. Nieużyte konstrukcje

- **Wątki (pthread)** - projekt oparty w całości na procesach
- **Łącza (pipe, mkfifo)** - zastąpione kolejkami komunikatów
- **Gniazda (socket)** - nie wymagane w projekcie