Serwis Samochodowy

Temat 6

Projekt zaliczeniowy z przedmiotu
Systemy Operacyjne

Semestr zimowy 2025/2026

Autor:

Martin Paszow
Nr albumu: 155209

Środowisko testowe: Ubuntu/Linux

Repozytorium: https://github.com/K0dziaK/serwis-samochodowy

2 lutego 2026

Spis treści

1 Wstęp

Niniejszy dokument stanowi raport z realizacji projektu zaliczeniowego z przedmiotu Systemy Operacyjne. Projekt realizuje temat nr 6 – **Serwis samochodowy**.

Celem projektu było stworzenie symulacji wieloprocesowej działania serwisu samochodowego z wykorzystaniem mechanizmów komunikacji międzyprocesowej (IPC) systemu Unix/Linux: kolejek komunikatów, pamięci dzielonej oraz semaforów.

2 Założenia projektowe

2.1 Opis problemu

W pewnej miejscowości znajduje się serwis samochodów dostępny w godzinach od T_p do T_k. Serwis obsługuje tylko samochody marek: A, E, I, O, U i Y (samogłoski alfabetu). Pozostałe marki – z zakresu od A do Z (łącznie 26 różnych marek) – nie są obsługiwane.

W serwisie znajduje się **8 stanowisk** do naprawy pojazdów:

- Stanowiska 1–7: obsługują marki A, E, I, O, U i Y
- Stanowisko 8: obsługuje tylko marki U i Y (specjalizacja)

2.2 Parametry konfiguracyjne

Parametr	**Opis**	**Wartość**
OPEN_HOUR	Godzina otwarcia serwisu	8:00
CLOSE_HOUR	Godzina zamknięcia serwisu	18:00
NUM_MECHANICS	Liczba mechaników	8
NUM_SERVICE_STAFF	Liczba stanowisk obsługi	3
K1	Próg uruchomienia stanowiska 2	3
K2	Próg uruchomienia stanowiska 3	5
T1	Max. czas oczekiwania poza godz. pracy	2h
CHANCE_RESIGNATION	Szansa rezygnacji klienta	2%
CHANCE_EXTRA_REPAIR	Szansa wykrycia dodatkowej usterki	20%
CHANCE_EXTRA_REPAIR_REFUSAL	Szansa odmowy dod. naprawy	20%
SIM_MINUTES_PER_REAL_SEC	Tempo symulacji	30 min/s

Tabela 1: Parametry konfiguracyjne symulacji

2.3 Reguły działania serwisu

2.3.1 Obsługa klientów

1. Serwis obsługuje tylko samochody marek: A, E, I, O, U i Y
2. Samochody przybywają losowo, nawet poza godzinami otwarcia
3. Jeżeli samochód przyjedzie poza godzinami pracy, może czekać w kolejce gdy:
    - Usterka jest krytyczna (3 typy napraw)

- Czas do otwarcia jest krótszy niż T_1 godzin

4. Pracownik serwisu określa przybliżony czas i koszt naprawy
5. Ok. 2% kierowców nie akceptuje warunków i odjeżdża
6. W ok. 20% przypadków wykrywane są dodatkowe usterki

2.3.2 Stanowiska obsługi klientów

- Zawsze działa minimum 1 stanowisko
- Drugie stanowisko uruchamia się gdy kolejka $> K_1$ (3 osoby)
- Trzecie stanowisko uruchamia się gdy kolejka $> K_2$ (5 osób)
- Stanowisko 2 zamyka się przy kolejce ≤ 2
- Stanowisko 3 zamyka się przy kolejce ≤ 3

2.3.3 Sygnały kierownika

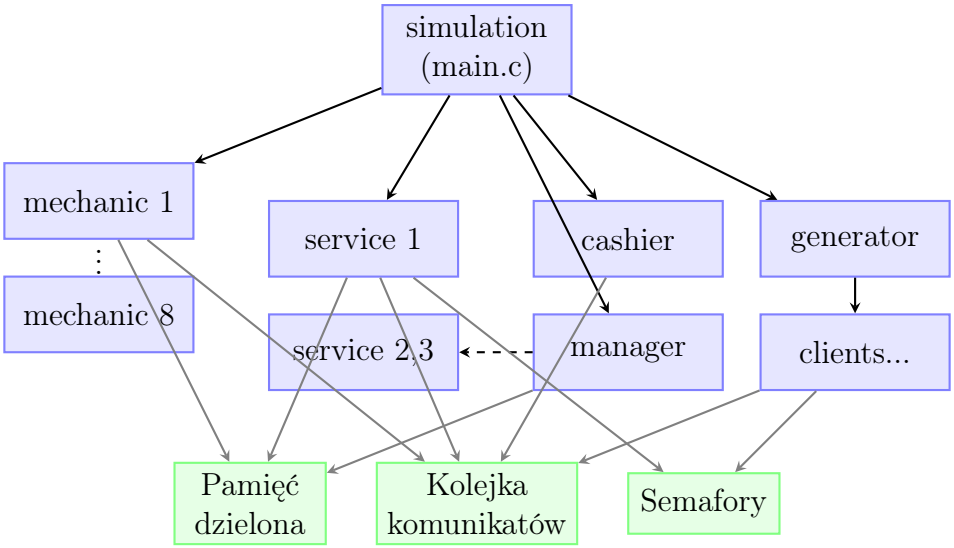
Sygnał	**Akcja**	**Implementacja**
1	Zamknięcie stanowiska mechanika	SIGUSR1
2	Przyspieszenie naprawy o 50%	SIGUSR2
3	Przywrócenie normalnej prędkości	SIGRTMIN
4	Pożar – ewakuacja serwisu	SIGTERM

Tabela 2: Sygnały sterujące pracą mechaników

3 Architektura systemu

3.1 Model wieloprocesowy

Projekt wykorzystuje architekturę wieloprocesową z komunikacją IPC. Każda rola (mechanik, obsługa, kasjer, klient, menedżer) działa jako osobny proces uruchamiany przez kombinację `fork()` + `exec()`.



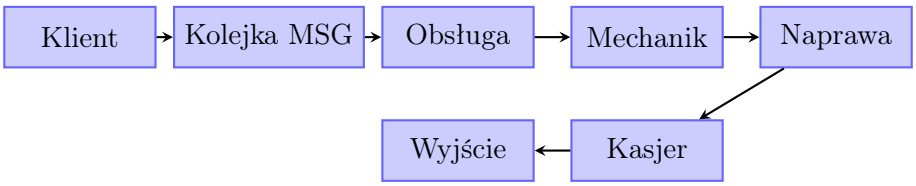
Rysunek 1: Diagram architektury procesów

3.2 Struktura plików źródłowych

Plik	Opis
`main.c`	Główny proces symulacji – inicjalizacja IPC, uruchamianie procesów potomnych, monitorowanie stanu
`common.h`	Nagłówek z definicjami struktur, stałych, typów komunikatów i deklaracjami funkcji
`common.c`	Implementacja funkcji pomocniczych: wrappery IPC, logowanie, cennik usług
`role_mechanic.c`	Logika procesu mechanika – obsługa napraw i sygnałów kierownika
`role_service.c`	Logika procesu obsługi klienta – przyjmowanie zgłoszeń, wycena, przypisywanie mechaników
`role_cashier.c`	Logika procesu kasjera – rozliczanie klientów
`role_client.c`	Logika procesu klienta – cały cykl życia klienta w serwisie
`role_manager.c`	Logika procesu menedżera – zarządzanie stanowiskami obsługi, sygnały
`client_generator.c`	Generator procesów klientów
`*_main.c`	Pliki wejściowe dla `exec()` poszczególnych ról
`Makefile`	Skrypt kompilacji

Tabela 3: Struktura plików projektu

3.3 Przepływ danych



Rysunek 2: Przepływ obsługi klienta

4 Mechanizmy komunikacji międzyprocesowej (IPC)

Projekt wykorzystuje trzy główne mechanizmy IPC systemu Unix/Linux:

4.1 Kolejki komunikatów (Message Queues)

Kolejki komunikatów stanowią **główny mechanizm komunikacji** między procesami. Wykorzystywana jest jedna kolejka z różnymi typami komunikatów (mtype).

4.1.1 Typy komunikatów

Stała	Wartość	Opis
MSG_CLIENT_TO_SERVICE	1	Zgłoszenie klienta do obsługi
MSG_MECHANIC_TO_SERVICE	2	Komunikat mechanika do obsługi
MSG_SERVICE_TO_CASHIER	3	Przekazanie klienta do kasy
MSG_BASE_TO_CLIENT	1000000 + PID	Wiadomość do konkretnego klienta
MSG_BASE_CLIENT_DECISION	2000000 + PID	Decyzja klienta
MSG_BASE_TO_MECHANIC	3000000 + ID	Zlecenie dla mechanika
MSG_BASE_CLIENT_PAYMENT	4000000 + PID	Płatność klienta

Tabela 4: Typy komunikatów w kolejce

4.1.2 Struktura komunikatu

```
typedef struct {
    long mtype;           // Typ komunikatu
    pid_t client_pid;     // PID klienta
    pid_t sender_pid;     // PID nadawcy
    int mechanic_id;      // ID mechanika
    char brand;           // Marka samochodu
    int service_id;       // ID uslugi z cennika
    int cost;             // Koszt naprawy
    int duration;         // Czas naprawy
    int is_extra_repair;  // Flaga: dodatkowa usterka
    int decision;         // Decyzja klienta (1=tak, 0=nie)
} msg_buf;
```

Listing 1: Struktura komunikatu msg_buf

4.1.3 Funkcje operujące na kolejce

- `msgget()` – tworzenie/pobranie kolejki
- `msgsnd()` – wysyłanie komunikatu (wrapper: `safe_msgsnd()`)
- `msgrcv()` – odbieranie komunikatu (wrappery: `safe_msgrcv_wait()`, `safe_msgrcv_nowait()`)
- `msgctl()` – usuwanie kolejki przy zamknięciu

4.2 Pamięć dzielona (Shared Memory)

Pamięć dzielona służy do przechowywania stanu symulacji dostępnego dla wszystkich procesów.

```
typedef struct {
    time_t start_time;              // Czas startu symulacji
    int simulation_running;         // Flaga: 1=dziala, 0=koniec
    int clients_in_queue;           // Liczba klientow w kolejce
    pid_t mechanics_pids[8];        // PID-y mechanikow
    int mechanic_status[8];         // Status: 0=wolny, 1=zajety, 2=
   zamkniety
    pid_t service_pids[3];          // PID-y stanowisk obslugi
} shared_data;
```

Listing 2: Struktura pamięci dzielonej shared_data

4.2.1 Funkcje operujące na pamięci dzielonej

- `shmget()` – tworzenie segmentu (wrapper: `safe_shmget()`)
- `shmat()` – dołączanie do przestrzeni adresowej (wrapper: `safe_shmat()`)
- `shmdt()` – odłączanie segmentu
- `shmctl()` – usuwanie przy zamknięciu

4.3 Semafory (Semaphores)

Semafory zapewniają synchronizację dostępu do zasobów współdzielonych.

Semafor	Wartość pocz.	Przeznaczenie
SEM_LOG	1	Synchronizacja logowania do pliku
SEM_QUEUE	100	Limit miejsc w kolejce klientów
SEM_PROC_LIMIT	5000	Limit procesów klientów
SEM_MECHANIC_ASSIGN	1	Sekcja krytyczna przypisywania mechaników

Tabela 5: Semafory i ich przeznaczenie

4.3.1 Funkcje semaforowe

```
void sem_lock(int semid, int sem_num) {
    struct sembuf sb = {sem_num, -1, 0};
    safe_semop(semid, &sb, 1);
}

void sem_unlock(int semid, int sem_num) {
    struct sembuf sb = {sem_num, 1, 0};
    safe_semop(semid, &sb, 1);
}
```

Listing 3: Implementacja operacji semaforowych

5 Obsługa sygnałów

System wykorzystuje sygnały do sterowania pracą mechaników i zarządzania symulacją.

5.1 Sygnały sterujące mechaników

```
void signal_handler(int sig) {
    if (sig == SIG_SPEED_UP) {
        if (speed_mode == 0) {
            speed_mode = 1;  // Przyspieszenie o 50%
        }
    }
    else if (sig == SIG_RESTORE_SPEED) {
        if (speed_mode == 1) {
            speed_mode = 0;  // Normalna predkosc
        }
    }
    else if (sig == SIG_CLOSE_STATION) {
        close_requested = 1;  // Zamkniecie po naprawie
    }
    else if (sig == SIG_FIRE) {
        exit(0);  // Natychmiastowa ewakuacja
    }
}
```

Listing 4: Handler sygnałów mechanika

5.2 Sygnały zarządzania symulacją

Sygnał	Handler	Akcja
SIGINT (Ctrl+C)	`sigint_handler`	Zakończenie symulacji, cleanup
SIGTSTP (Ctrl+Z)	`sigtstp_handler`	Wstrzymanie wszystkich procesów
SIGCONT	`sigcont_handler`	Wznowienie symulacji

Tabela 6: Sygnały zarządzania symulacją

6 Opis procesów

6.1 Proces główny (main.c)

Proces główny odpowiada za:

1. Inicjalizację zasobów IPC (kolejka, pamięć dzielona, semafory)
2. Uruchomienie procesów potomnych poprzez `fork()` + `exec()`
3. Monitorowanie stanu symulacji w głównej pętli
4. Wykrywanie awarii procesów potomnych (`waitpid()` z `WNOHANG`)
5. Zwalnianie zasobów przy zamknięciu (`cleanup()`)

6.2 Mechanik (role_mechanic.c)

Mechanik wykonuje naprawy samochodów:

- Czeka na zlecenie od obsługi (MSG_BASE_TO_MECHANIC + ID)
- Symuluje naprawę (pierwsza połowa czasu)
- Losowo wykrywa dodatkowe usterki (20% szans)
- Komunikuje się z obsługą w sprawie dodatkowych napraw
- Kończy naprawę i przekazuje formularz do obsługi
- Obsługuje sygnały kierownika (przyspieszenie, zamknięcie)

6.3 Obsługa klienta (role_service.c)

Stanowisko obsługi:

- Przyjmuje zgłoszenia od klientów
- Weryfikuje markę samochodu (tylko A, E, I, O, U, Y)
- Wycenia naprawę na podstawie cennika (±10%)
- Czeka na decyzję klienta
- Przypisuje mechanika (sekcja krytyczna z semaforem)
- Obsługuje komunikaty od mechaników (dodatkowe usterki, zakończenie)
- Przekazuje gotowych klientów do kasjera

6.4 Kasjer (role_cashier.c)

Kasjer obsługuje płatności:

- Czeka na komunikaty od obsługi (MSG_SERVICE_TO_CASHIER)
- Wystawia rachunek klientowi
- Czeka na płatność (MSG_BASE_CLIENT_PAYMENT)
- Wydaje kluczyki

6.5 Klient (role_client.c)

Proces klienta przechodzi przez cały cykl obsługi:

1. Losowanie marki (A-Z) i usterki z cennika
2. Decyzja o wejściu do kolejki (poza godzinami: tylko usterki krytyczne)
3. Wysłanie zgłoszenia do obsługi
4. Oczekiwanie na ofertę
5. Decyzja o akceptacji/odrzuceniu (2% odrzuceń)
6. Oczekiwanie podczas naprawy
7. Decyzja o dodatkowych naprawach (20% odmów)
8. Płatność u kasjera
9. Opuszczenie serwisu

6.6 Generator klientów (client_generator.c)

Generator tworzy nowych klientów:

- Działa w pętli dopóki symulacja trwa
- Pobiera slot z semafora SEM_PROC_LIMIT
- Tworzy proces klienta przez `fork()`
- Wywołuje `run_client()` w procesie potomnym
- Losowe opóźnienie między kolejnymi klientami

6.7 Menedżer (role_manager.c)

Menedżer zarządza serwisem:

- Monitoruje długość kolejki
- Uruchamia/zamyka stanowiska 2 i 3 dynamicznie
- Przyjmuje polecenia z klawiatury (1-4)
- Wysyła sygnały do mechaników

7 Cennik usług

ID	Nazwa usługi	Koszt	Czas	Kryt.
	Usługi eksploatacyjne			
0	Wymiana oleju i filtrów	300 PLN	2	Nie
1	Wymiana opon (sezonowa)	150 PLN	2	Nie
2	Wymiana żarówki	50 PLN	1	Nie
3	Wymiana wycieraczek	80 PLN	1	Nie
4	Przegląd klimatyzacji	200 PLN	3	Nie
5	Mycie i woskowanie	120 PLN	2	Nie
6	Wymiana płynu spryskiwaczy	30 PLN	1	Nie
7	Diagnostyka komputerowa	150 PLN	2	Nie
8	Wymiana filtra powietrza	60 PLN	1	Nie
9	Wymiana filtra kabinowego	70 PLN	1	Nie
	Naprawy mechaniczne			
10	Wymiana klocków hamulcowych	400 PLN	4	**Tak**
11	Wymiana tarcz hamulcowych	600 PLN	5	**Tak**
12	Wymiana amortyzatorów	800 PLN	6	Nie
13	Wymiana świec zapłonowych	250 PLN	3	Nie
14	Naprawa tłumika	350 PLN	4	Nie
15	Wymiana akumulatora	400 PLN	1	**Tak**
16	Regulacja zbieżności kół	200 PLN	3	Nie
17	Wymiana paska osprzętu	300 PLN	4	Nie
18	Naprawa zamka centralnego	250 PLN	3	Nie
19	Wymiana płynu chłodniczego	180 PLN	3	Nie
	Poważne awarie (krytyczne)			
20	Awaria układu kierowniczego	1500 PLN	10	**Tak**
21	Zerwany pasek rozrządu	2500 PLN	15	**Tak**
22	Wyciek paliwa	600 PLN	5	**Tak**
23	Awaria pompy hamulcowej	900 PLN	6	**Tak**
24	Przegrzewanie silnika	3000 PLN	20	**Tak**
25	Awaria skrzyni biegów	4000 PLN	18	**Tak**
26	Wymiana sprzęgła	1800 PLN	12	**Tak**
27	Naprawa alternatora	500 PLN	5	**Tak**
28	Naprawa rozrusznika	450 PLN	5	**Tak**
29	Pęknięta sprężyna zawieszenia	400 PLN	4	**Tak**

Tabela 7: Cennik usług serwisowych

8 Napotkane problemy i rozwiązania

8.1 Problem 1: Wyścigi przy zamykaniu symulacji

Opis: Przy zamykaniu symulacji procesy klientów próbowały wysyłać wiadomości do już usuniętych kolejek.

Rozwiązanie: Implementacja bezpiecznych wrapperów (`safe_msgrcv_wait`, `safe_msgsnd`)

sprawdzających kody błędów `EIDRM` i `EINVAL`. W przypadku usunięcia kolejki proces kończy się gracefully przez `exit(0)`.

8.2 Problem 2: Deadlock przy obsłudze dodatkowych usterek

Opis: Mechanik czekał na odpowiedź od obsługi, która czekała na odpowiedź od klienta – powstał cykliczny deadlock.

Rozwiązanie: Wprowadzenie nieblokującego odbierania wiadomości (`IPC_NOWAIT`) dla priorytetowych komunikatów od mechaników. Obsługa najpierw sprawdza komunikaty od mechaników, potem od nowych klientów.

8.3 Problem 3: Procesy Zombie

Opis: Procesy klientów stawały się zombie po zakończeniu, bo żaden proces ich nie "odbierał".

Rozwiązanie: Dodanie `waitpid(-1, &status, WNOHANG)` w pętli generatora oraz prawidłowe zwalnianie semaforów w funkcji `client_exit()`.

8.4 Problem 4: Race-condition przy przypisywaniu mechaników

Opis: Gdy program działał bez opóźnień (tryb `TESTING_NO_SLEEP`), występowały wyścigi gdzie mechanik dostawał zlecenia zanim zdążył zaktualizować swój status w pamięci dzielonej.

Rozwiązanie: Wprowadzono semafor binarny `SEM_MECHANIC_ASSIGN` synchronizujący sekcję krytyczną przypisywania mechaników. Status mechanika jest natychmiastowo aktualizowany na zajęty przed wysłaniem zlecenia.

```
sem_lock(sem_id, SEM_MECHANIC_ASSIGN);
for (int i = NUM_MECHANICS - 1; i >= 0; i--) {
    if (shm->mechanic_status[i] == 0) {
        found = i + 1;
        shm->mechanic_status[i] = 1; // Natychmiast zajety
        break;
    }
}
sem_unlock(sem_id, SEM_MECHANIC_ASSIGN);
```

Listing 5: Sekcja krytyczna przypisywania mechanika

8.5 Problem 5: Ujemna liczba klientów w kolejce

Opis: Licznik `clients_in_queue` osiągał wartości ujemne przez wielokrotną dekrementację.

Rozwiązanie: Przebudowano system odłączania klientów. Funkcja `client_exit()` przyjmuje flagę `in_queue` określającą czy wymagana jest dekrementacja. Dodatkowo sprawdzany jest warunek `if (shm->clients_in_queue > 0)`.

8.6 Problem 6: Awaria procesu nie kończyła symulacji

Opis: Jeśli któryś z procesów potomnych kończył się z błędem, pozostałe nadal działały w nieprawidłowym stanie.

Rozwiązanie: W głównej pętli dodano skanowanie procesów przy użyciu `waitpid(-1, &status, WNOHANG)`. Dla każdego zakończonego procesu krytycznego sprawdzany jest kod wyjścia – jeśli niezerowy, ustawiana jest flaga `simulation_running = 0`.

9 Zrealizowane funkcjonalności

9.1 Wymagania podstawowe

- ✓ Obsługa tylko marek A, E, I, O, U, Y
- ✓ 8 stanowisk naprawczych (stanowisko 8 tylko dla U i Y)
- ✓ Godziny otwarcia T_p–T_k
- ✓ Klienci przychodzą losowo (nawet poza godzinami)
- ✓ Oczekiwanie poza godzinami dla usterek krytycznych lub gdy $T < T_1$
- ✓ Cennik z 30 usługami
- ✓ Akceptacja/odrzucenie warunków przez klienta (∼2% odrzuceń)
- ✓ Wykrywanie dodatkowych usterek (∼20% przypadków)
- ✓ Dynamiczne stanowiska obsługi (1–3) w zależności od kolejki
- ✓ Sygnały kierownika (1–4)

9.2 Wymagania dodatkowe

- ✓ Walidacja danych wejściowych
- ✓ Obsługa błędów z `perror()` i `errno`
- ✓ Minimalne prawa dostępu do struktur IPC (0666)
- ✓ Usuwanie struktur IPC po zakończeniu
- ✓ Logowanie akcji do pliku `raport.txt`

9.3 Elementy wyróżniające

1. **Kolorowane wyjście terminala** – czytelne logowanie z timestampami i kolorami dla każdej roli
2. **Tryb testowy bez sleep** – makro `TESTING_NO_SLEEP` dla szybkiego testowania
3. **Pause/Resume symulacji** – obsługa Ctrl+Z (SIGTSTP) i wznowienia (SIGCONT)
4. **Dynamiczne zarządzanie procesami obsługi** przez menedżera
5. **Kompletny cennik usług** z podziałem na kategorie
6. **Wykrywanie awarii procesów potomnych** z automatycznym zamknięciem symulacji

10 Testy

Poniżej przedstawiono szczegółowy opis scenariuszy testowych przeprowadzonych w celu weryfikacji poprawności działania systemu.

Test 1: Obsługa nieobsługiwanych marek
Cel: Sprawdzenie czy serwis prawidłowo odrzuca marki spoza zbioru {A, E, I, O, U, Y}.
Procedura: Uruchomienie symulacji i obserwacja logów dla klientów z wylosowanymi markami takimi jak B, C, D itp.
Oczekiwany rezultat: Klienci z nieobsługiwanymi markami są informowani o braku możliwości obsługi i natychmiast opuszczają serwis.
Status: ✓ ZALICZONY

Test 2: Dynamiczne uruchamianie stanowisk obsługi
Cel: Sprawdzenie czy stanowiska 2 i 3 są uruchamiane oraz zamykane zgodnie z progami K_1 i K_2.
Procedura: Zwiększenie częstotliwości generowania klientów i obserwacja logów menedżera.
Oczekiwany rezultat: Pojawienie się komunikatów „MANAGER: Uruchamiam Obsługę 2/3" w momencie, gdy długość kolejki przekroczy K_1 lub K_2.
Status: ✓ ZALICZONY

Test 3: Sygnał pożaru (sygnał 4)
Cel: Sprawdzenie natychmiastowego zamknięcia serwisu.
Procedura: Wprowadzenie cyfry „4" w terminalu podczas działania symulacji.
Oczekiwany rezultat: Wszystkie procesy (mechanicy, obsługa, klienci) kończą pracę, a zasoby IPC są zwalniane.
Status: ✓ ZALICZONY

Test 4: Przyspieszenie i przywrócenie prędkości naprawy
Cel: Sprawdzenie obsługi sygnałów 2 i 3 dla konkretnego mechanika.
Procedura: Wprowadzenie polecenia „2 1" (przyspieszenie mechanika nr 1), a następnie „3 1" (przywrócenie normalnej prędkości).
Oczekiwany rezultat: Czas naprawy zostaje skrócony o 50%, a po drugim sygnale wraca do wartości bazowej.
Status: ✓ ZALICZONY

Test 5: Zamknięcie stanowiska mechanika (sygnał 1)
Cel: Sprawdzenie czy mechanik kończy bieżącą naprawę przed zamknięciem stanowiska.
Procedura: Wprowadzenie polecenia „1 3" podczas gdy mechanik nr 3 wykonuje pracę.
Oczekiwany rezultat: Mechanik nie przerywa pracy natychmiast, lecz wyświetla komunikat: „Mechanik 3: Zamykam stanowisko po zleceniu".
Status: ✓ ZALICZONY

Test 6: Obsługa poza godzinami otwarcia (usterka krytyczna)
Cel: Sprawdzenie czy klienci z usterkami krytycznymi czekają w kolejce przed otwarciem serwisu.

Procedura: Uruchomienie symulacji przed godziną `OPEN_HOUR` **Oczekiwany rezultat:** Klient nie odjeżdża, lecz czeka w kolejce do momentu otwarcia serwisu (godziny 8:00) - kolejka rośnie.
Status: ✓ ZALICZONY

Test 7: Działanie programu bez opóźnień (stress test)
Cel: Sprawdzenie stabilności programu pod dużym obciążeniem bez funkcji `sleep()`.
Procedura: Odkomentowanie makra `#define TESTING_NO_SLEEP` w pliku nagłówkowym i rekompilacja.
Oczekiwany rezultat: Program działa stabilnie przez dłuższy czas, nie występują wyścigi ani zakleszczenia.
Status: ✓ ZALICZONY

Test 8: Wykrywanie awarii procesu potomnego
Cel: Sprawdzenie czy awaria (SIGKILL) jednego z procesów kończy całą symulację.
Procedura: Ręczne zabicie procesu z kodem błędu: `kill -9 <PID_mechanika>` z innego terminala.
Oczekiwany rezultat: Główny proces wykrywa zmianę stanu, wyświetla komunikat „[MAIN] Wykryto nieautoryzowane zabicie procesu" i bezpiecznie zamyka symulację.
Status: ✓ ZALICZONY

Test 9: Specjalizacja mechanika 8
Cel: Sprawdzenie czy mechanik nr 8 obsługuje wyłącznie marki U i Y.
Procedura: Obserwacja logów przypisań mechaników dla różnych marek samochodów.
Oczekiwany rezultat: Mechanik 8 otrzymuje zlecenia tylko dla marek U i Y, podczas gdy pozostałe marki trafiają wyłącznie do mechaników 1–7.
Status: ✓ ZALICZONY

Test 10: Odrzucenie oferty przez klienta
Cel: Sprawdzenie obsługi przypadku, gdy klient rezygnuje z naprawy po otrzymaniu wyceny.
Procedura: Uruchomienie symulacji i obserwacja logów (szansa rezygnacji wynosi ok. 2%).
Oczekiwany rezultat: Komunikat „Klient X odrzucił koszty", po czym klient opuszcza serwis bez angażowania mechanika.
Status: ✓ ZALICZONY

Test 11: Brak wolnych mechaników
Cel: Sprawdzenie obsługi sytuacji, gdy wszystkie stanowiska mechaników są zajęte.
Procedura: Zwiększenie obciążenia serwisu lub ręczne wyłączenie kilku mechaników sygnałem 1.
Oczekiwany rezultat: Komunikat „BRAK WOLNYCH MIEJSC. Odprawiam klienta X". Klient jest kierowany do kasy z kosztem 0 PLN.
Status: ✓ ZALICZONY

Test 12: Wykrywanie dodatkowych usterek
Cel: Sprawdzenie mechanizmu wykrywania i obsługi dodatkowych usterek w trakcie trwa-

nia naprawy.
Procedura: Obserwacja logów mechaników (szansa wykrycia wynosi ok. 20%).
Oczekiwany rezultat: Wyświetlenie komunikatu o wykryciu dodatkowej usterki, zapytanie klienta o zgodę oraz kontynuacja lub zakończenie naprawy w zależności od decyzji klienta.
Status: ✓ ZALICZONY

11 Linki do istotnych fragmentów kodu

Poniżej znajdują się bezpośrednie odnośniki do implementacji kluczowych mechanizmów w repozytorium GitHub.

11.1 Tworzenie i obsługa plików

- `fopen()`, `fprintf()`, `fclose()` – logowanie do raport.txt

11.2 Tworzenie procesów

- `fork()` – tworzenie procesów mechaników
- `execl()` – uruchamianie programów ról
- `exit()` – zakończenie procesu
- `wait()`/`waitpid()` – oczekiwanie na procesy potomne

11.3 Obsługa sygnałów

- `signal()` – ustawienie handlerów
- `sigaction()` – zaawansowana obsługa sygnałów mechanika
- `kill()` – wysyłanie sygnałów do procesów
- `raise()` – wysłanie sygnału do siebie

11.4 Synchronizacja procesów (semafory)

- `ftok()` – generowanie klucza IPC
- `semget()` – tworzenie zestawu semaforów
- `semctl()` – inicjalizacja semaforów
- `semop()` – operacje na semaforach (lock/unlock)

11.5 Segmenty pamięci dzielonej

- `shmget()` – tworzenie segmentu
- `shmat()` – dołączanie segmentu
- `shmdt()` – odłączanie segmentu
- `shmctl()` – usuwanie segmentu

11.6 Kolejki komunikatów

- `msgget()` – tworzenie kolejki
- `msgsnd()` – wysyłanie wiadomości
- `msgrcv()` – odbieranie wiadomości
- `msgctl()` – usuwanie kolejki

12 Kompilacja i uruchomienie

12.1 Kompilacja

```
make clean
make
```

12.2 Uruchomienie

```
./simulation
```

12.3 Sterowanie podczas działania

Polecenie	Akcja
`1 <id_mech>`	Zamknięcie stanowiska mechanika
`2 <id_mech>`	Przyspieszenie naprawy o 50%
`3 <id_mech>`	Przywrócenie normalnej prędkości
`4`	Pożar (natychmiastowe zamknięcie)
`Ctrl+C`	Normalne zamknięcie symulacji
`Ctrl+Z`	Pauza symulacji
`fg`	Wznowienie symulacji

Tabela 8: Polecenia sterujące

12.4 Czyszczenie

```
make clean
```

13 Podsumowanie

Projekt realizuje wszystkie wymagania tematu 6 – Serwis samochodowy. Zaimplementowano pełną symulację wieloprocesową z wykorzystaniem mechanizmów IPC:

- **Kolejki komunikatów** – główny mechanizm komunikacji między procesami
- **Pamięć dzielona** – przechowywanie stanu symulacji
- **Semafory** – synchronizacja dostępu do zasobów
- **Sygnały** – sterowanie pracą mechaników i zarządzanie symulacją

System jest odporny na błędy, prawidłowo zwalnia zasoby przy zamykaniu i loguje wszystkie akcje do pliku `raport.txt`. Architektura wieloprocesowa z wykorzystaniem `fork()` + `exec()` zapewnia izolację poszczególnych ról i ułatwia debugowanie.