POLITECHNIKA WROCŁAWSKA
WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

Politechnika
Wrocławska

Symulacja lotniska przy wykorzystaniu procesów

Przedmiot: Systemy Operacyjne - projekt
Prowadzący: dr inż. Dominik Żelazny
Termin: PIĄTEK TP 15:15

Autor:
Przemysław Dyjak 280959

WROCŁAW, 2026

Spis treści

Rozdział 1

Wstęp

1.1 Cel projektu

Celem projektu było zaprojektowanie oraz implementacja symulacji złożonego systemu zarządzania zasobami z wykorzystaniem mechanizmów współbieżności na poziomie procesów w systemie operacyjnym Linux. Jako temat przewodni symulacji wybrano funkcjonowanie portu lotniczego. Jest to klasyczny problem informatyczny, w którym występuje konieczność synchronizacji dostępu wielu niezależnych podmiotów – samolotów – do ograniczonych zasobów współdzielonych, takich jak pasy startowe, bramki, cysterny paliwowe, ekipy techniczne i inne. Projekt został zrealizowany w języku C++ z wykorzystaniem natywnego API systemowego POSIX. Kluczowym założeniem projektu było oparcie architektury na wieloprocesowości, a nie wielowątkowości, co wymusiło zastosowanie zaawansowanych mechanizmów komunikacji międzyprocesowej IPC - **Inter-Process Communication** - takich jak pamięć współdzielona oraz semafory. Aplikacja wizualizuje stan lotniska w czasie rzeczywistym, obsługując dynamiczne pojawianie się nowych samolotów, obsługę pasażerów oraz sytuacje losowe, jak np. brak paliwa w zbiorniku lotniskowym.

1.2 Proces w systemie operacyjnym

Fundamentalną jednostką wykonawczą w zrealizowanym projekcie jest proces. W terminologii systemów operacyjnych proces definiuje się jako instancję wykonywanego programu. Jest to dynamiczny byt, który posiada własny licznik rozkazów, zawartość rejestrów procesora oraz – co istotne – własną, odseparowaną wirtualną przestrzeń adresową. Każdy proces w systemie Linux jest identyfikowany przez unikalny numer **PID**. System operacyjny zarządza procesami poprzez strukturę danych zwaną blokiem kontrolnym procesu - **PCB** - która przechowuje informacje o stanie procesu - np. działający, gotowy, oczekujący - priorytecie, otwartych deskryptorach plików oraz uprawnieniach. W kontekście tego projektu, każdy samolot jest odrębnym procesem, co symuluje niezależność decyzyjną każdej maszyny.

1.3 Porównanie procesów i wątków

Wybór architektury wieloprocesowej zamiast wielowątkowej niesie ze sobą istotne implikacje programistyczne i systemowe. Podstawowe różnice obejmują:

Przestrzeń adresowa i izolacja
: Procesy są od siebie silnie odizolowane. Błąd w jednym procesie zazwyczaj nie wpływa bezpośrednio na stabilność innych procesów. Wymaga to jednak jawnego stosowania mechanizmów IPC - np. pamięci współdzielonej - do wymiany danych. Wątki współdzielą tę samą przestrzeń adresową. Komunikacja jest prostsza, ale błąd jednego wątku może spowodować awarię całego programu.

Koszt tworzenia i przełączania
: Proces jest tzw. jednostką ciężką. Jego utworzenie wymaga skopiowania tablic stron pamięci i struktur systemowych, co jest operacją bardziej kosztowną obliczeniowo. Wątek jest jednostką lekką, a jego narzut systemowy jest znacznie mniejszy.

Zastosowanie modelu wieloprocesowego pozwoliło na wierne odwzorowanie środowiska, w którym każdy samolot funkcjonuje jako w pełni autonomiczny byt systemowy. Architektura ta nie tylko lepiej oddaje specyfikę rzeczywistych systemów rozproszonych, ale również w pełni realizuje założenia projektowe zawarte w poleceniu.

1.4 Cykl życia procesu

W systemach rodziny Unix zarządzanie cyklem życia procesu opiera się na specyficznych wywołaniach systemowych, które zostały zaimplementowane w projekcie:

- **Tworzenie procesu – fork -** nowy proces powstaje poprzez wywołanie funkcji systemowej `fork()`. Tworzy ona nowy proces o identycznym obrazie pamięci, zazwyczaj z wykorzystaniem mechanizmu *Copy-on-Write*. Proces potomny otrzymuje nowy PID, a jego pamięć jest kopią pamięci rodzica.
- **Uruchamianie nowego programu – exec -** po wykonaniu `fork()`, proces potomny zazwyczaj wywołuje funkcję z rodziny `exec`. Powoduje to zastąpienie obrazu pamięci procesu potomnego nowym kodem programu resetując stos i stertę, ale zachowując PID.
- **Kończenie procesu – exit -** proces kończy działanie dobrowolnie wywołując `exit()` lub w wyniku sygnału z zewnątrz. Wówczas zwalnia on swoje zasoby, ale jego wpis w tablicy procesów pozostaje do momentu odczytania statusu zakończenia przez rodzica.
- **Problem procesów Zombie -** proces, który zakończył działanie, ale nie został odebrany przez rodzica, staje się tzw. procesem Zombie. W projekcie zastosowano funkcję, aby proces główny - rodzic - na bieżąco odbierał statusy zakończenia samolotów, zapobiegając wyciekom zasobów systemowych.

Rozdział 2

Podejście do problemu

2.1 Komunikacja między procesami

Jako mechanizm wymiany danych wybrano **pamięć współdzieloną POSIX**. Wszystkie procesy mapują ten sam obszar pamięci RAM, w którym znajduje się struktura pamięci współdzielonej. Dzięki temu zmiany stanu – np. zajęcie pasa startowego przez jeden proces – są natychmiast widoczne dla wszystkich pozostałych procesów oraz dla modułu rysującego interfejs, bez narzutu czasowego na kopiowanie danych.

2.2 Synchronizacja i zarządzanie zasobami

Aby zapobiec sytuacjom wyścigu - *race conditions* - przy dostępie do wspólnej pamięci, zastosowano semafory POSIX:

- **Ochrona danych -** semafory binarne chroniące spójność danych podczas zapisu i odczytu.
- **Zarządzanie pulą zasobów -** semafory licznikowe dla zasobów przenoszonych - np. ekipy naziemne. Proces wykonujący `sem_wait` na takim semaforze jest usypiany przez system, jeśli licznik wynosi 0, co oznacza aktualny brak zasobu.

Dla zasobów takich jak pas startowy i bramka gate zastosowano podejście aktywnego sprawdzania, co pozwala samolotowi szukać alternatywnego zasobu zamiast blokować się na pierwszym zajętym.

Rozdział 3

Opis działania i implementacja

Rozdział 3 zawiera szczegółową analizę techniczną zaimplementowanego rozwiązania. System symulacji lotniska został zaprojektowany jako aplikacja wieloprocesowa, działająca w środowisku systemu operacyjnego Linux. Architektura rozwiązania opiera się na ścisłej separacji procesu zarządczego - lotniska - oraz procesów roboczych - samolotów - komunikujących się wyłącznie poprzez mechanizmy IPC zgodne ze standardem POSIX.

3.1 Struktura pamięci współdzielonej i IPC

Fundamentem działania systemu jest obiekt pamięci współdzielonej, tworzony za pomocą wywołania systemowego `shm_open` oraz mapowany do przestrzeni adresowej procesów funkcją `mmap`. Współdzielony obszar pamięci pełni rolę centralnej bazy danych dla symulacji i zawiera:

- **Zmienne konfiguracyjne:** wartości wprowadzone przez użytkownika przy starcie - liczba pasów, bramek, ekip itd. - które determinują limity pętli w procesach potomnych.
- **Tablice semaforów:**
  - `sem_t runways[]` – tablica semaforów binarnych reprezentująca pasy startowe.
  - `sem_t gates[].lock` – semafory binarne chroniące dostęp do poszczególnych bramek.
- **Semafory licznikowe:** reprezentujące pule zasobów przenośnych - `ground_crews`, `fuel_trucks`, `deicers`. Ich wartość początkowa odpowiada liczbie dostępnych jednostek sprzętu.
- **Semafor warunkowy:** `fuel_delivery_wait` – specyficzny semafor zainicjowany na wartość 0, służący do usypiania procesów w oczekiwaniu na dostawę paliwa.
- **Tablice struktur danych:** tablica `planes` przechowująca stan każdego aktywnego samolotu - PID, status, liczba pasażerów - oraz tablica `terminal_queue` reprezentująca pasażerów oczekujących w terminalu.

Dostęp do całej struktury - a w szczególności do danych wizualnych - jest dodatkowo synchronizowany globalnym semaforem `ui_lock`, co zapobiega efektom migotania interfejsu lub odczytu niepełnych danych przez wątek rysujący.

3.2 Faza inicjalizacji i konfiguracji dynamicznej

Program rozpoczyna działanie od interaktywnej fazy konfiguracyjnej w trybie tekstowym. Jest to kluczowy etap, w którym następuje alokacja zasobów logicznych.

Algorytm przydziału bramek

W przeciwieństwie do statycznej konfiguracji, zaimplementowano kaskadowy system przydziału bramek z puli głównej - stała `MAX_GATES`.

1. Użytkownik deklaruje liczbę bramek małych (S). System weryfikuje, czy wartość nie przekracza `MAX_GATES`.
2. Pozostała liczba bramek - `gates_left` - staje się górnym limitem dla bramek średnich (M).
3. Analogicznie, reszta puli zostaje udostępniona dla bramek dużych (L).

Taki mechanizm zapobiega błędom segmentacji i logicznym niespójnościom, gwarantując, że suma bramek poszczególnych typów nigdy nie przekroczy rozmiaru tablicy zaalokowanej w pamięci współdzielonej.

Mapowanie i inicjalizacja zasobów

Po zatwierdzeniu konfiguracji, proces główny tworzy segment pamięci współdzielonej. Następuje kluczowy moment inicjalizacji semaforów POSIX. Algorytm inicjalizacji jest dynamiczny – pętle `for` iterują wyłącznie do wartości podanych przez użytkownika, a nie do stałych technicznych tablicy. Dzięki temu, mimo że tablica `runways` może pomieścić technicznie 10 semaforów, jeśli użytkownik wybrał 2 pasy, tylko 2 pierwsze semafory zostaną zainicjowane funkcją `sem_init` i będą brały udział w symulacji.

3.3 Proces zarządcy - lotnisko

Główny proces aplikacji pełni rolę orkiestratora - schedulera - oraz interfejsu użytkownika. Jego działanie opiera się na nieskończonej pętli `while(running)`, w której każda iteracja realizuje szereg zadań administracyjnych.

Obsługa wejścia i tworzenie procesów

Zastosowano nieblokującą obsługę klawiatury. Wciśnięcie klawisza 'p' inicjuje procedurę narodzin nowego samolotu:

1. Przeszukiwana jest tablica `planes` w pamięci współdzielonej w poszukiwaniu wolnego slotu, gdzie `pid == 0`.
2. Po znalezieniu slotu następuje wywołanie funkcji systemowej `fork()`.
3. **W procesie rodzica -** PID nowego dziecka jest zapisywany w slocie, a status ustawiany na `ARRIVING`.
4. **W procesie dziecka -** następuje wywołanie `execv`, które podmienia kod procesu na plik wykonywalny `./plane_main`. Jako argumenty wywołania przekazywane są ID slotu oraz ziarno losowości. Jest to niezbędne, aby nowy proces wiedział, którym elementem tablicy w pamięci współdzielonej ma zarządzać.

Symulator terminala pasażerskiego

W każdej klatce symulacji istnieje prawdopodobieństwo 15% pojawienia się nowej grupy pasażerów. Grupy te - liczące losowo 1, 3, 5, 7 lub 9 osób - trafiają do tablicy `terminal_queue`. Równocześnie realizowany jest algorytm niecierpliwości. Dla każdej aktywnej grupy obliczana jest różnica czasu:

$$\Delta t = \texttt{time(NULL)} - \texttt{arrival\_time}$$

Jeżeli $\Delta t >$ `PASSENGER_PATIENCE_SEC`, grupa jest usuwana z kolejki, a licznik rezygnacji `pax_resigned` jest inkrementowany. Operacje te są wykonywane w sekcji krytycznej chronionej semaforem `ui_lock`.

Problem producenta - dostawy paliwa

Aby zasymulować realne ograniczenia zasobów, zaimplementowano mechanizm zużywalnego paliwa. Proces główny pełni rolę producenta paliwa. Co 20 sekund sprawdzany jest poziom w zbiorniku głównym `fuel_stock`. Jeżeli poziom jest niski, następuje dostawa i zwiększenie wartości zmiennej. Kluczowym elementem jest tutaj obsługa procesów oczekujących. Samoloty, którym zabrakło paliwa, śpią na semaforze `fuel_delivery_wait`. Proces główny po dolaniu paliwa sprawdza licznik `planes_waiting_for_delivery` i wykonuje pętlę wywołań `sem_post`, budząc dokładnie tylu konsumentów, ilu czekało na surowiec.

Eliminacja procesów Zombie

W systemach Unix proces potomny po zakończeniu działania staje się tzw. Zombie do momentu odebrania jego kodu wyjścia przez rodzica. Aby uniknąć wycieku zasobów systemowych, w głównej pętli zaimplementowano mechanizm asynchronicznego sprzątania - Listing 3.1.

Listing 3.1: Asynchroniczne usuwanie procesów Zombie

```
1 while(waitpid(-1, NULL, WNOHANG) > 0);
```

Flaga `WNOHANG` sprawia, że funkcja nie blokuje działania symulacji, jeśli żaden samolot nie zakończył pracy, ale natychmiastowo sprząta te, które odleciały.

3.4 Proces samolotu – szczegóły implementacji

Logika biznesowa samolotu została wydzielona do osobnego pliku `plane_main.cpp`. Każdy samolot jest niezależnym bytem, który po uruchomieniu podłącza się do istniejącego segmentu pamięci współdzielonej. Cykl życia samolotu to maszyna stanów, w której przejścia zależą od dostępności zasobów.

Lądowanie i aktywne oczekiwanie

W fazie lądowania samolot musi zająć jeden z pasów startowych. Ze względu na to, że pasów może być wiele, nie można użyć prostego `sem_wait` na jednym semaforze. Zastosowano algorytm *polling*:

1. Proces iteruje po wszystkich dostępnych pasach - od 0 do `config_runways`.
2. Dla każdego pasa wywoływane jest `sem_trywait`. Funkcja ta próbuje opuścić semafor, ale w przypadku jego zajęcia nie blokuje procesu, lecz zwraca błąd.
3. Jeśli uda się zająć pas - zwrócono 0 - samolot ląduje.
4. Jeśli żaden pas nie jest wolny, proces usypia na krótki czas - `usleep` - i ponawia próbę.

Jest to kompromis między wydajnością a prostotą implementacji obsługi wielu równorzędnych zasobów.

Logika doboru bramki gate

Po wylądowaniu następuje procedura wyboru bramki. Jest ona bardziej złożona niż wybór pasa, ponieważ bramki mają atrybut rozmiaru - S, M, L. Zastosowano algorytm dopasowania *Best Fit/Any Fit*:

- Samolot rozmiaru S akceptuje bramki S, M oraz L.
- Samolot rozmiaru M akceptuje bramki M oraz L.
- Samolot rozmiaru L akceptuje wyłącznie bramki L.

Proces iteruje po tablicy bramek, sprawdzając zarówno kompatybilność rozmiaru, jak i dostępność semafora. Po sukcesie, ID bramki jest zapisywane w strukturze samolotu, co pozwala na jej wizualizację w interfejsie graficznym.

Boarding pasażerów

Proces boardingu symuluje interakcję między dwoma strukturami danych w pamięci współdzielonej. Samolot skanuje tablicę `terminal_queue`. Jeśli znajdzie grupę pasażerów, której cel podróży pokrywa się z celem lotu samolotu, oraz w samolocie jest wystarczająco dużo miejsca, grupa jest oznaczana jako usunięta z terminala, a licznik pasażerów na pokładzie wzrasta. Czas trwania boardingu jest dynamicznie skalowany w zależności od liczby zabranych pasażerów.

Obsługa naziemna i problem konsumenta

Etap obsługi naziemnej wymaga synchronizacji z trzema typami zasobów.

Ekipy techniczne i odladzarki

Dla ekip - `ground_crews` - i odladzarek - `deicers` - wykorzystano semafory licznikowe. Proces wywołuje `sem_trywait`. Jeśli zasób nie jest dostępny - licznik semafora = 0 - proces wchodzi w stan oczekiwania `WAIT_CREW` lub `WAIT_DEICER` i cyklicznie ponawia próbę.

Tankowanie – złożona synchronizacja

Procedura tankowania jest najbardziej złożonym elementem logicznym, implementującym problem producenta i konsumenta z dodatkowym zasobem wyłącznym, jakim jest cysterna.

1. **Zajęcie cysterny -** samolot najpierw musi pozyskać cysternę.
2. **Sprawdzenie stanu paliwa -** proces wchodzi w sekcję krytyczną chronioną semaforem `fuel_stock_lock`, aby odczytać globalny stan paliwa.
3. **Oczekiwanie warunkowe -** jeśli paliwa jest za mało:
   - Proces inkrementuje licznik oczekujących - `planes_waiting_for_delivery`.
   - Zwalnia blokadę zbiornika - `sem_post(&fuel_stock_lock)` - aby producent mógł dolać paliwa.
   - Usypia się na semaforze prywatnym `fuel_delivery_wait`.
4. **Wznowienie -** po obudzeniu przez producenta, proces ponownie zajmuje `fuel_stock_lock`, pobiera paliwo, zwalnia blokadę i przechodzi do symulacji fizycznego tankowania.

3.5 Procedura startu i zwalnianie zasobów

Po zakończeniu obsługi, samolot zwalnia bramkę - podnosząc semafor `gates[i].lock`. Następnie procedura jest analogiczna do lądowania – samolot musi uzyskać wyłączny dostęp do pasa startowego. Po symulacji startu, proces aktualizuje globalne statystyki - liczba obsłużonych lotów, pasażerów, zużyte paliwo - w pamięci współdzielonej. Ostatnim krokiem jest wywołanie `exit(0)`. Powoduje to wysłanie sygnału `SIGCHLD` do procesu rodzica, który - dzięki wspomnianej pętli `waitpid` - usunie wpis z tablicy procesów systemowych.

3.6 Zakończenie i sprzątanie

Poprawne zakończenie symulacji jest kluczowe dla stabilności systemu operacyjnego. Ponieważ używamy mechanizmów IPC, nie są one automatycznie usuwane po zabiciu procesu - tzw. *kernel persistence*. W funkcji `cleanup_ipc`:

- Wszystkie semafory są niszczone.
- Pamięć jest odmapowywana.
- Obiekt pamięci współdzielonej jest usuwany z systemu plików.

Dodatkowo, proces główny iteruje po liście aktywnych samolotów i wysyła im sygnał `SIGKILL`, aby upewnić się, że po zamknięciu terminala w tle nie pozostaną żadne osierocone procesy robocze.

Rozdział 4

Najważniejsze wstawki kodu źródłowego

Rozdział 4 prezentuje kluczowe fragmenty kodu źródłowego, które realizują fundamentalne założenia projektu - współdzielenie pamięci, zarządzanie procesami oraz synchronizację dostępu do zasobów.

4.1 Struktura pamięci współdzielonej

Struktura `AirportSharedState`, której definicję przedstawiono na Listingu 4.1, stanowi serce systemu. Jest ona mapowana przez wszystkie procesy i zawiera zarówno konfigurację, semafory POSIX, jak i dynamiczne dane symulacji.

Listing 4.1: Definicja struktury pamięci współdzielonej - shared_state.h

```
struct AirportSharedState {

    sem_t ui_lock;

    // Konfiguracja i limity
    int config_runways;
    int config_gates_S;
    int config_gates_M;
    int config_gates_L;
    int config_total_gates; // Suma S+M+L
    // ... (pozostałe zmienne konfiguracyjne)

    // ZASOBY STALE
    sem_t runways[MAX_RUNWAYS];     // Tablica semaforów dla pasów
    pid_t runway_pid[MAX_RUNWAYS];  // Kto zajmuje pas - pid
    Gate gates[MAX_GATES];          // Tablica bramek

    // ZASOBY PRZENOSZONE
    sem_t ground_crews; // Semafor zliczający ekipy
    sem_t fuel_trucks;  // Semafor zliczający cysterny
    sem_t deicers;      // Semafor zliczający odladzarki

    // ZASOB ZUZYWALNY - synchronizacja warunkowa
    sem_t fuel_stock_lock;
    long long fuel_stock;            // Ile litrów w zbiorniku
    sem_t fuel_delivery_wait;        // Dla samolotów bez paliwa
    int planes_waiting_for_delivery; // Ilu śpi w poczekalni

    // DANE DLA UI
    PlaneSlot planes[MAX_CONCURRENT_PLANES];
    int planes_spawned_count;

    // Terminal z pasażerami
    Passenger terminal_queue[MAX_WAITING_PASSENGERS];

    // Statystyki
    Statistics stats;
};
```

4.2 Inicjalizacja środowiska - proces rodzic

Funkcja `initialize_state` - Listingi 4.2 oraz 4.3 - odpowiada za utworzenie obiektu pamięci współdzielonej w systemie, zmapowanie go oraz inicjalizację semaforów zgodnie z limitami podanymi przez użytkownika.

Listing 4.2: Tworzenie SHM i dynamiczna inicjalizacja semaforów - airport_main.cpp - cz. 1

```
// Inicjalizacja całego świata pamięci i semaforów
void initialize_state(UserConfig cfg) { // przyjmuje config
    //  plik pamięci współdzielonej
    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(AirportSharedState));

    // mapowanie go do pamięci tego procesu
    void* ptr = mmap(NULL, sizeof(AirportSharedState), PROT_READ |
        ↪ PROT_WRITE, MAP_SHARED, shm_fd, 0);
    state = static_cast<AirportSharedState*>(ptr);

    // czyszczenie pamięci zerami
    memset(state, 0, sizeof(AirportSharedState));

    // ZAPISANIE KONFIGURACJI DO PAMIECI WSPOLDZIELONEJ
    state->config_runways = cfg.runways;
    // ... (pozostałe)

    sem_init(&state->ui_lock, 1, 1);

    // Pasy startowe - tylko tyle ile w konfigu
    for(int i=0; i<state->config_runways; ++i) sem_init(&state->
        ↪ runways[i], 1, 1);

    // Bramki rozmiary - dynamiczne
    int gid = 0;
    // małe S
    for(int i=0; i<state->config_gates_S; ++i) { state->gates[gid].
        ↪ size = PSize::S; sem_init(&state->gates[gid++].lock, 1, 1);
        ↪  }
    // srednie M
    for(int i=0; i<state->config_gates_M; ++i) { state->gates[gid].
        ↪ size = PSize::M; sem_init(&state->gates[gid++].lock, 1, 1);
        ↪  }
    // duże L
    for(int i=0; i<state->config_gates_L; ++i) { state->gates[gid].
        ↪ size = PSize::L; sem_init(&state->gates[gid++].lock, 1, 1);
        ↪  }
```

Listing 4.3: Tworzenie SHM i dynamiczna inicjalizacja semaforów - airport_main.cpp - cz. 2

```
    // Pule zasobów - semafor zlicza ile wolnych wg konfigu
    sem_init(&state->ground_crews, 1, state->config_crews);
    sem_init(&state->fuel_trucks, 1, state->config_trucks);
    sem_init(&state->deicers, 1, state->config_deicers);

    // Paliwo
    sem_init(&state->fuel_stock_lock, 1, 1);
    state->fuel_stock = INITIAL_FUEL_STOCK;
    sem_init(&state->fuel_delivery_wait, 1, 0); // Startuje zamknięty
        ↪  - nikt nie czeka
}
```

4.3 Zarządzanie procesami

Kluczowy mechanizm symulacji obejmuje dwa etapy - tworzenie nowego, niezależnego procesu samolotu funkcją `spawn_plane` - Listing 4.4 - oraz podłączanie się procesu dziecka do istniejącej pamięci funkcją `attach_memory` - Listing 4.5.

Listing 4.4: Tworzenie procesu potomnego fork/exec - airport_main.cpp

```
1  void spawn_plane() {
2      sem_wait(&state->ui_lock);
3      // Znajduje wolny slot w tablicy
4      int slot_id = -1;
5      for(int i=0; i<MAX_CONCURRENT_PLANES; ++i) {
6          if (state->planes[i].pid == 0) { slot_id = i; break; }
7      }
8      if (slot_id == -1) { sem_post(&state->ui_lock); return; } // Brak
           ↪  miejsca
9
10     // FORK tworzy nowy proces
11     pid_t pid = fork();
12
13     if (pid == 0) {
14         // W PROCESIE DZIECKA
15         // Zastępuje ten proces programem plane_main
16         std::string s_id = std::to_string(slot_id);
17         std::string s_rnd = std::to_string(state->
               ↪ planes_spawned_count);
18         char* args[] = { (char*)"./plane_main", (char*)s_id.c_str(),
               ↪ (char*)s_rnd.c_str(), NULL };
19         execv(args[0], args);
20         exit(1); // Jeśli execv zawiedzie
21     }
22     else {
23         // w procesie rodzicu
24         state->planes[slot_id].pid = pid;
25         state->planes[slot_id].status = PStatus::ARRIVING;
26         state->planes[slot_id].passengers_on_board = 0; // Reset
27         state->planes_spawned_count++;
28         //state->stats.total_planes++; // Statystyka
29         sem_post(&state->ui_lock);
30     }
31 }
```

Listing 4.5: Podłączanie procesu dziecka do SHM - plane_main.cpp

```
// funkcja podłączająca proces do pamięci współdzielonej IPC
void attach_memory() {
    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);

    // mapuje pamięć do przestrzeni adresowej tego procesu
    void* ptr = mmap(NULL, sizeof(AirportSharedState), PROT_READ |
        ↪ PROT_WRITE, MAP_SHARED, shm_fd, 0);

    // rzut na moja strukturę
    state = static_cast<AirportSharedState*>(ptr);
    close(shm_fd);
}
```

4.4 Logika samolotu - synchronizacja zasobów

Samoloty rywalizują o zasoby. Listing 4.6 przedstawia implementację aktywnego odpytywania o wolny pas startowy, co zapobiega trwałemu zablokowaniu procesu na jednym, zajętym semaforze.

Listing 4.6: Aktywne oczekiwanie na pas startowy - plane_main.cpp

```
// Zajmowanie pasa startowego
int acquire_runway() {
    int id = -1;
    // Pętla pollingu
    while (id == -1) {
        for (int i = 0; i < state->config_runways; ++i) { //używa
            ↪ konfigu
            // sem_trywait sprawdzi czy wolny bez blokowania
            if (sem_trywait(&state->runways[i]) == 0) {
                id = i; // Udało się zająć

                // Zapisze PID żeby UI widziało kto zajął
                sem_wait(&state->ui_lock);
                state->runway_pid[i] = getpid();
                sem_post(&state->ui_lock);
                break;
            }
        }
        if (id == -1) usleep(100000);
        // Jak zajęte, śpi 0.1s i próbuje znowu
    }
    return id;
}
```

4.5 Problem producenta i konsumenta - paliwo

Najbardziej złożony mechanizm synchronizacji w projekcie. Listing 4.7 obrazuje konsumenta, który w razie braku paliwa zwalnia blokadę i usypia się. Listing 4.8 pokazuje producenta, który po dolaniu paliwa budzi oczekujące procesy.

Listing 4.7: Konsument paliwa z oczekiwaniem warunkowym - plane_main.cpp

```
void process_refueling() {
        int needed = 7000 + (rand() % 5000); // Ile paliwa potrzeba

        while (sem_trywait(&state->fuel_trucks) != 0) {
            set_status(PStatus::WAIT_FUEL_TRUCK);
            work_ms(500, 1500);
        }
        set_status(PStatus::REFUELING); // Ma cysternę

        // blokuje zbiornik
        sem_wait(&state->fuel_stock_lock);

        // sprawdza stan zbiornika
        while (state->fuel_stock < needed) {
            // Za mało paliwa
            set_status(PStatus::WAIT_FUEL_DELIVERY);
            state->planes_waiting_for_delivery++;

            // zwalnia zbiornik żeby dostawca mógł dolać
            sem_post(&state->fuel_stock_lock);

            // sleep
            sem_wait(&state->fuel_delivery_wait);

            // Obudzono  -> znowu blokowanie zbiornika
            sem_wait(&state->fuel_stock_lock);
        }
        // pobranie paliwa
        state->fuel_stock -= needed;
        state->stats.total_fuel += needed; // Statystyka
        // Zwalnia zbiornik dla innych
        sem_post(&state->fuel_stock_lock);

        set_status(PStatus::REFUELING);
        work_ms(1000, 6000); // Fizyczne tankowanie
        // Oddaje cysternę
        sem_post(&state->fuel_trucks);
}
```

Listing 4.8: Producent paliwa - budzenie procesów - airport_main.cpp

```
// LOGIKA PRODUCENTA PALIWA
void run_fuel_producer() {
    // blokada zbiornika
    sem_wait(&state->fuel_stock_lock);

    // warunek dolewania
    if (state->fuel_stock <= 80000) {

        state->fuel_stock += FUEL_REFILL_AMOUNT; // Dolewa

        // skoro dolane to budzi wszystkich czekających
        int waiters = state->planes_waiting_for_delivery;
        state->planes_waiting_for_delivery = 0;

        sem_post(&state->fuel_stock_lock); // Odblokowuje zbiornik

        // Wysyła sygnał pobudki
        for(int i=0; i<waiters; ++i) {
            sem_post(&state->fuel_delivery_wait);
        }
    }
    else {
        // jeśli paliwa duzo to nic nie robi ale oddac klodke trzeba
        sem_post(&state->fuel_stock_lock);
    }
}
```

4.6 Logika biznesowa i sprzątanie

Przykład bezpiecznej interakcji z danymi w pamięci współdzielonej widoczny jest na Listingu 4.9. Z kolei Listing 4.10 prezentuje funkcję czyszczącą, która usuwa semafory i odmapowuje pamięć przy zamykaniu programu.

Listing 4.9: Boarding pasażerów w sekcji krytycznej - plane_main.cpp

```
// boarding pasazerow
void perform_boarding() {
    set_status(PStatus::BOARDING);
    work_ms(1000, 2000);

    sem_wait(&state->ui_lock);

    int taken = 0;
    // Skan kolejki pasażerów w terminalu
    for(int i=0; i<MAX_WAITING_PASSENGERS; ++i) {
        // Jeśli pasażer istnieje i chce lecieć tam gdzie samolot
        if (state->terminal_queue[i].is_active && state->
            ↪ terminal_queue[i].dest == my_slot->dest) {
            // Sprawdzam czy cała grupa wejdzie
            int grp_size = state->terminal_queue[i].group_size;

            // Jeśli jest w samolocie jeszcze miejsce dla calej grupy
            if (taken + grp_size <= my_slot->capacity) {
                state->terminal_queue[i].is_active = false; //
                    ↪ zabieram pasazerow
                taken += grp_size; // dodaje liczebnosc grupy
            }
            // Jeśli nie, grupa zostaje i czeka na nastepny
        }
    }
    my_slot->passengers_on_board = taken; // Zapisze ilu wzial
       ↪ samolot

    sem_post(&state->ui_lock); // Koniec sekcji krytycznej

    // Symulacja czasu wchodzenia zależy od liczby pasażerów
    work_ms(taken * 100, taken * 200 + 500);
}
```

Listing 4.10: Sprzątanie semaforów i pamięci - airport_main.cpp

```
// Sprzątanie pamięci współdzielonej i semaforów
void cleanup_ipc() {
    if (state != nullptr) {
        sem_destroy(&state->ui_lock);
        // Pętle zależne od konfiguracji
        for(int i=0; i<state->config_runways; ++i) sem_destroy(&state
            ↪ ->runways[i]);
        for(int i=0; i<state->config_total_gates; ++i) sem_destroy(&
            ↪ state->gates[i].lock);
        sem_destroy(&state->ground_crews);
        sem_destroy(&state->fuel_trucks);
        sem_destroy(&state->deicers);
        sem_destroy(&state->fuel_stock_lock);
        sem_destroy(&state->fuel_delivery_wait);
        munmap(state, sizeof(AirportSharedState));
    }
    if (shm_fd != -1) close(shm_fd);
    shm_unlink(SHM_NAME); // Kasuje plik pamięci z systemu
}
```

Rozdział 5

Symulacja i testy

W celu weryfikacji poprawności działania systemu oraz zbadania wydajności mechanizmów synchronizacji przeprowadzono serię testów programu. Każdy test polegał na uruchomieniu symulacji, skonfigurowaniu środowiska według zadanych parametrów oraz obsłużeniu stałej liczby 10 samolotów. Samoloty były generowane w krótkich i zawsze **identycznych** odstępach czasu.

5.1 Konfiguracja środowiska

Po uruchomieniu programu, proces główny przechodzi w tryb interaktywnej konfiguracji. Użytkownik definiuje limity zasobów, które determinują stopień trudności obsługi ruchu lotniczego - Listing 5.1

Listing 5.1: Konfiguracja

```
KONFIGURACJA LOTNISKA
Podaj liczbe pasow startowych (1 - 4): ... wpisz liczbę

KONFIGURACJA BRAMEK - lacznie maks 20
Zostalo do dyspozycji: 20 bramek.
Podaj liczbe malych bramek (S): ... wpisz liczbę
Zostalo do dyspozycji: X bramek.
Podaj liczbe srednich bramek (M): ... wpisz liczbę
Zostalo do dyspozycji: Y bramek.
Podaj liczbe duzych bramek (L): ... wpisz liczbę

ZASOBY OBSLUGI
Podaj liczbe ekip technicznych (1 - 20): ... wpisz liczbę
Podaj liczbe cystern z paliwem (1 - 20): ... wpisz liczbę
Podaj liczbe odladzarek (1 - 20): ... wpisz liczbę
```

5.2 Scenariusz 1 - deficyt zasobów - wąskie gardło

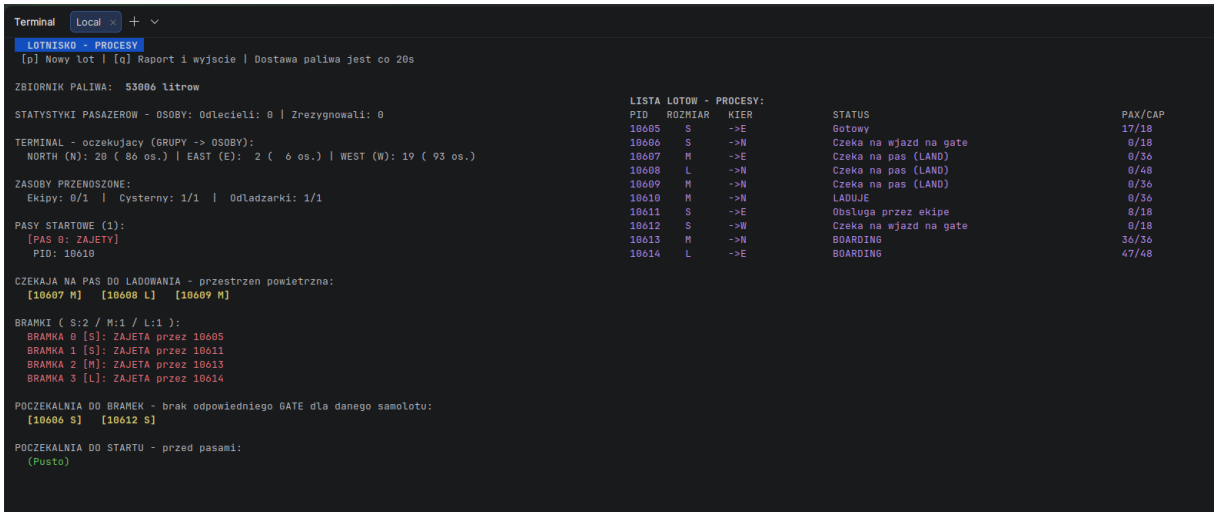
Celem tego testu było sprawdzenie zachowania systemu w warunkach skrajnego niedoboru zasobów. Oczekiwano powstania długich kolejek czekania na pas startowy - w powietrzu i na ziemi - oraz blokowania bramek z powodu braku obsługi naziemnej. Wiąże się to również z wysokim poziomem rezygnacji oczekujących grup pasażerów.

Parametry testu:

Tab. 5.1: Konfiguracja dla scenariusza deficytowego

Zasób	Wartość
Pasy startowe	1
Bramki	2 S, 1 M, 1 L
Ekipy techniczne	1
Cysterny paliwowe	1
Odladzarki	1

Zgodnie z przewidywaniami, pojedynczy pas startowy stał się krytycznym wąskim gardłem. Samoloty długo oczekiwały na lądowanie i start. Dodatkowo, posiadanie tylko jednej ekipy technicznej i jednej cysterny sprawiło, że obsługa naziemna odbywała się sekwencyjnie – samoloty zajmowały bramki znacznie dłużej niż wynikałoby to z samej procedury serwisowej, blokując miejsce dla kolejnych maszyn. Przebieg tego scenariusza testowego przedstawia Rysunek 5.1. Raport statystyk przedstawiono na Rysunku 5.2.



Rys. 5.1: Screen z początku symulacji - scenariusz deficytowy

```
przemyslaw_dyjak@przemek-ASUS-TUF-Gaming-A15-FA507NU-FA507NU:~/Pulpit/ProjektyCLion/ProcesyLotnisko/build$ ./airport_main

KONFIGURACJA LOTNISKA
Podaj liczbe pasow startowych (1 - 4): 1

KONFIGURACJA BRAMEK - lacznie maks 20
Zostalo do dyspozycji: 20 bramek.
Podaj liczbe malych bramek (S): 2
Zostalo do dyspozycji: 18 bramek.
Podaj liczbe srednich bramek (M): 1
Zostalo do dyspozycji: 17 bramek.
Podaj liczbe duzych bramek (L): 1

ZASOBY OBSLUGI
Podaj liczbe ekip technicznych (1 - 20): 1
Podaj liczbe cystern z paliwem (1 - 20): 1
Podaj liczbe odladzarek (1 - 20): 1




======================================================================
                  RAPORT KONCOWY SYMULACJI LOTNISKA
======================================================================
 1. RUCH LOTNICZY:
    - Liczba lacznie obsluzonych samolotow, ktore odlecialy: 10
       - Typ S: 4
       - Typ M: 4
       - Typ L: 2
    - Zuzyte paliwo:  90373 litrow
    - Odladzanie:     3 operacji

 2. STATYSTYKI PASAZEROW (OSOBY):
    - Pojawilo sie w terminalu:                              1174
    - Zrezygnowalo przez timeout:                            650
    - Lacznie odlecialo:                                     298
    - W terminalu i samolotach, ktore nie odlecialy zostalo:    226

 3. SREDNIE OBICIAZENIE:
    - Srednia liczba pasazerow na lot - ogolnie: 29.80
    - Srednio wg typu samolotu:
       > Typ S (max 18):  15.25 pax/lot
       > Typ M (max 36): 36.00 pax/lot
       > Typ L (max 48): 46.50 pax/lot
    - Srednio wg kierunku lotu:
       > NORTH: 34.40 pax/lot (lotow: 5)
       > EAST:  27.00 pax/lot (lotow: 4)
       > WEST:  18.00 pax/lot (lotow: 1)
======================================================================
```

Rys. 5.2: Statystyki - scenariusz deficytowy

5.3 Scenariusz 2 - zasoby zrównoważone

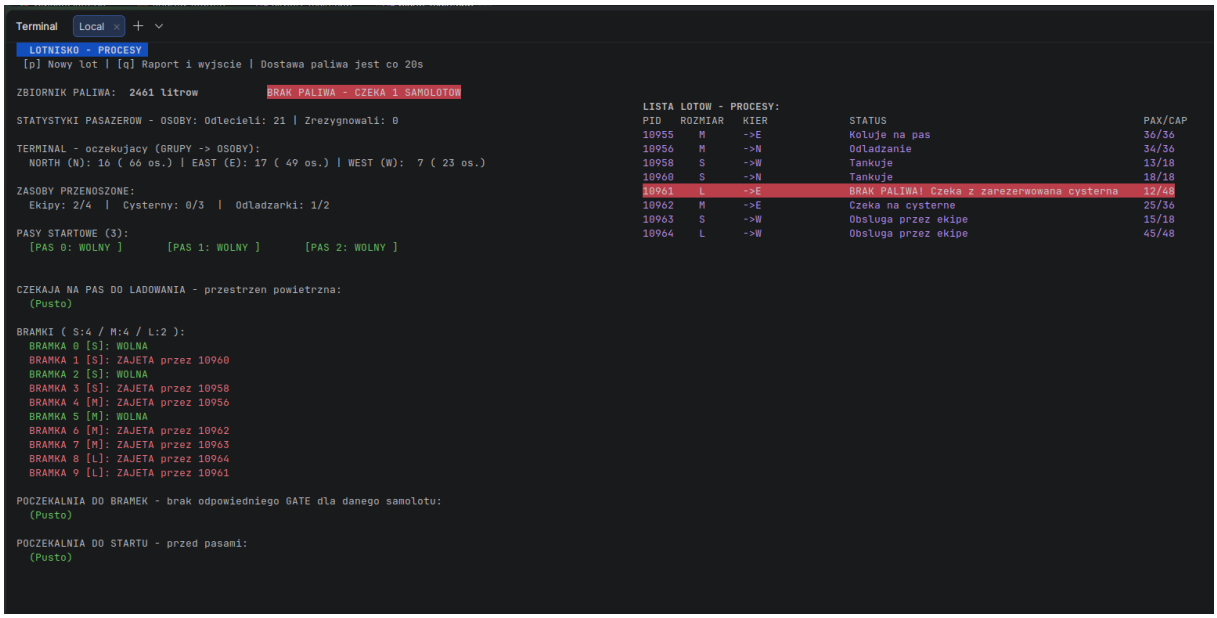
Scenariusz ten symuluje typowy dzień pracy lotniska, gdzie liczba zasobów jest dopasowana do przewidywanego ruchu, ale w momentach szczytowych mogą wystąpić chwilowe oczekiwania na zasoby i rezygnacje ze strony pasażerów.

Parametry testu:

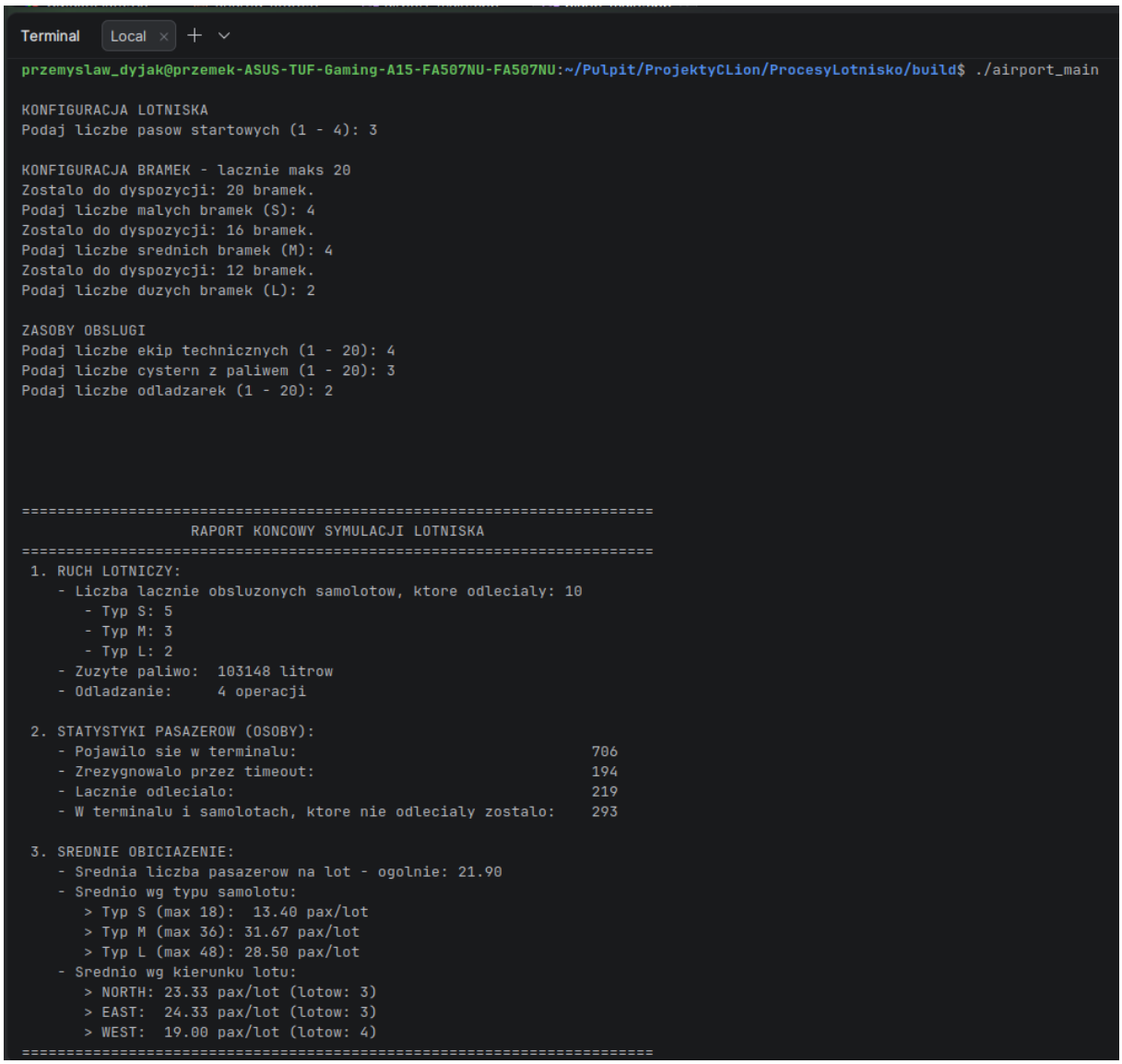
Tab. 5.2: Konfiguracja dla scenariusza zrównoważonego

Zasób	Wartość
Pasy startowe	3
Bramki	4 S, 4 M, 2 L
Ekipy techniczne	4
Cysterny paliwowe	3
Odladzarki	2

Dzięki zwiększeniu liczby pasów startowych, ruch odbywał się znacznie płynniej. Samoloty lądowały i startowały na trzech pasach. Zwiększona liczba bramek oraz ekip technicznych pozwoliła na równoległą obsługę kilku maszyn jednocześnie. Kolejki pojawiały się rzadziej - głównie przy kumulacji samolotów o tym samym rozmiarze - np. kilka samolotów L rywalizujących o bramki. Przebieg tego scenariusza testowego przedstawia Rysunek 5.3. Raport statystyk przedstawiono na Rysunku 5.4.



Rys. 5.3: Screen z początku symulacji - scenariusz zrównoważony



Rys. 5.4: Statystyki - scenariusz zrównoważony

5.4 Scenariusz 3 - nadmiar zasobów - maksymalna przepustowość

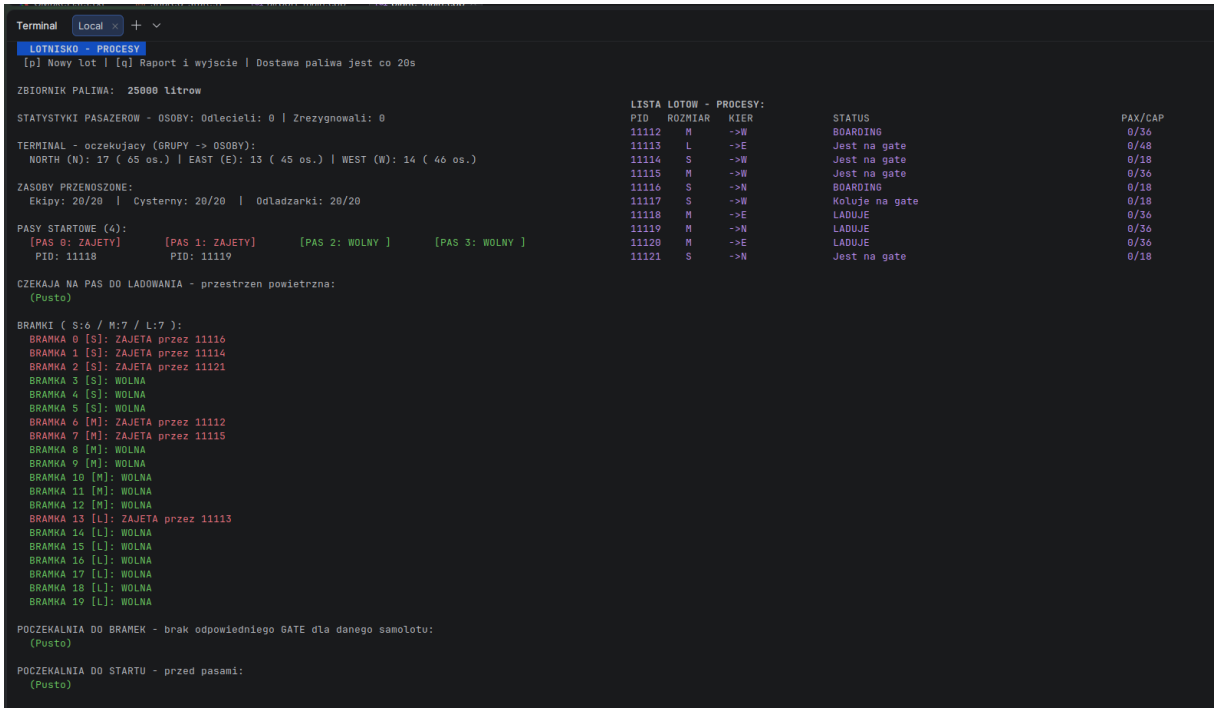
Test ten miał na celu sprawdzenie maksymalnej wydajności programu przy wykorzystaniu pełnych limitów technicznych zdefiniowanych w kodzie źródłowym. Zasoby przydzielono tak, aby żaden samolot nie musiał oczekiwać na obsługę.

Parametry testu:

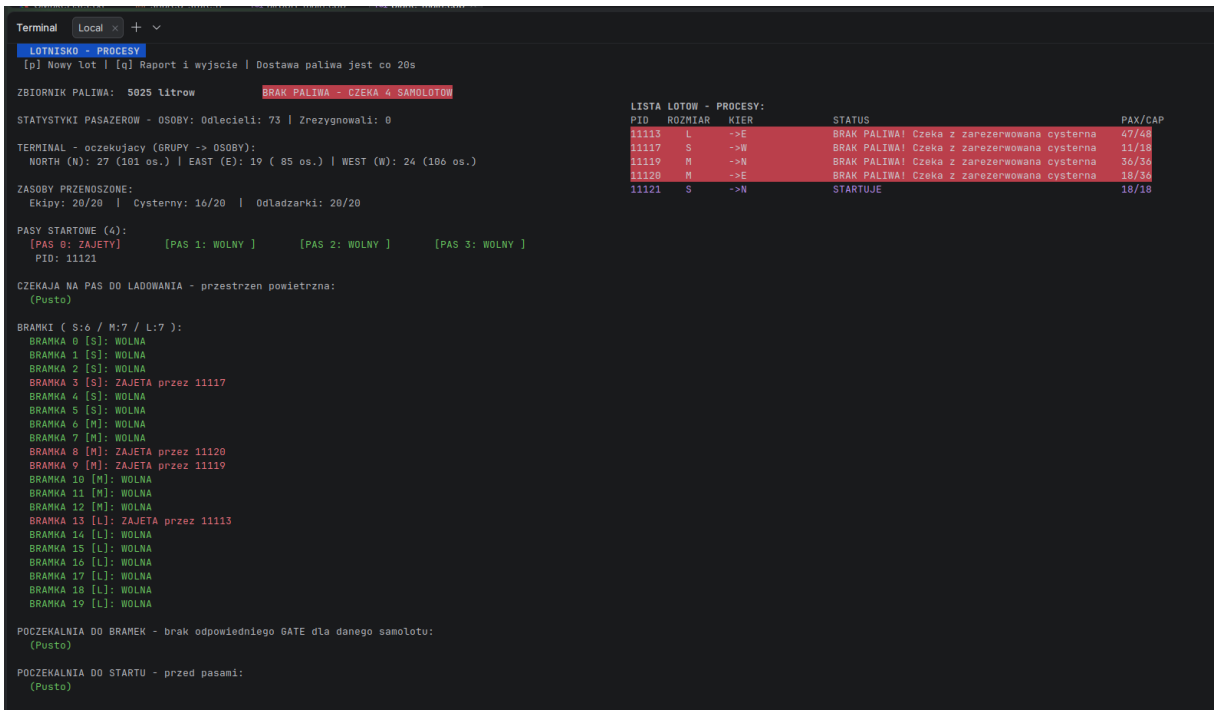
Tab. 5.3: Konfiguracja dla scenariusza maksymalnego

Zasób	Wartość
Pasy startowe	4 - MAX
Bramki	6 S, 7 M, 7 L
Ekipy techniczne	20
Cysterny paliwowe	20
Odladzarki	20

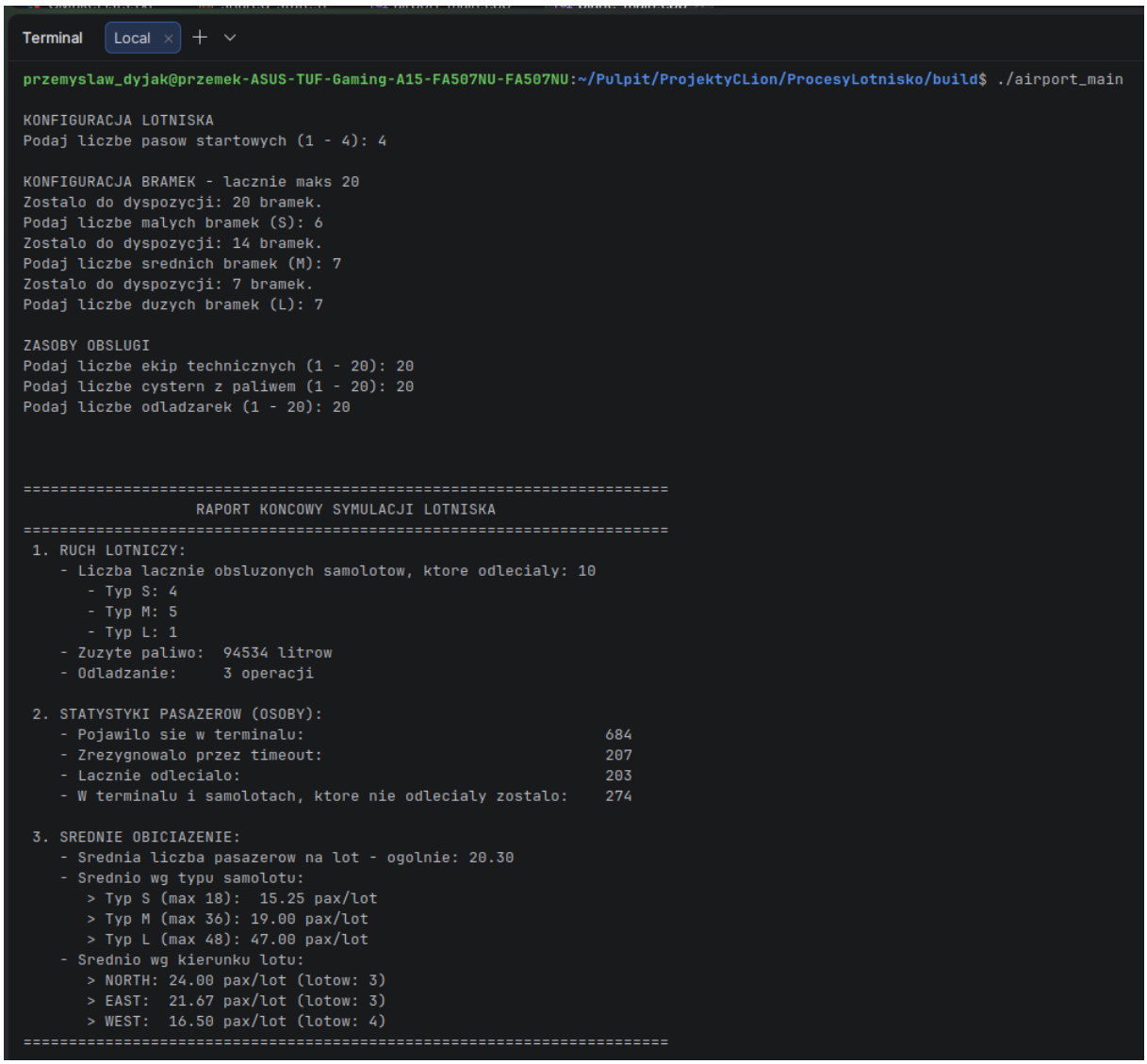
Przy czterech pasach startowych i dużej liczbie bramek i zasobów, ograniczeniem stał się jedynie czas trwania samych czynności i dostępność paliwa. Samoloty były obsługiwane natychmiast po zgłoszeniu zapotrzebowania. Jest to sytuacja prawie idealna, rzadko spotykana w rzeczywistości, ale potwierdzająca poprawność działania mechanizmów zwalniania semaforów - brak wycieków i zakleszczeń przy dużym obciążeniu. Przebieg tego scenariusza testowego przedstawia Rysunek 5.5. Oczekiwanie związane z brakiem paliwa przedstawia Rysunek 5.6. Raport statystyk przedstawiono na Rysunku 5.7.



Rys. 5.5: Screen z początku symulacji - scenariusz maksymalny

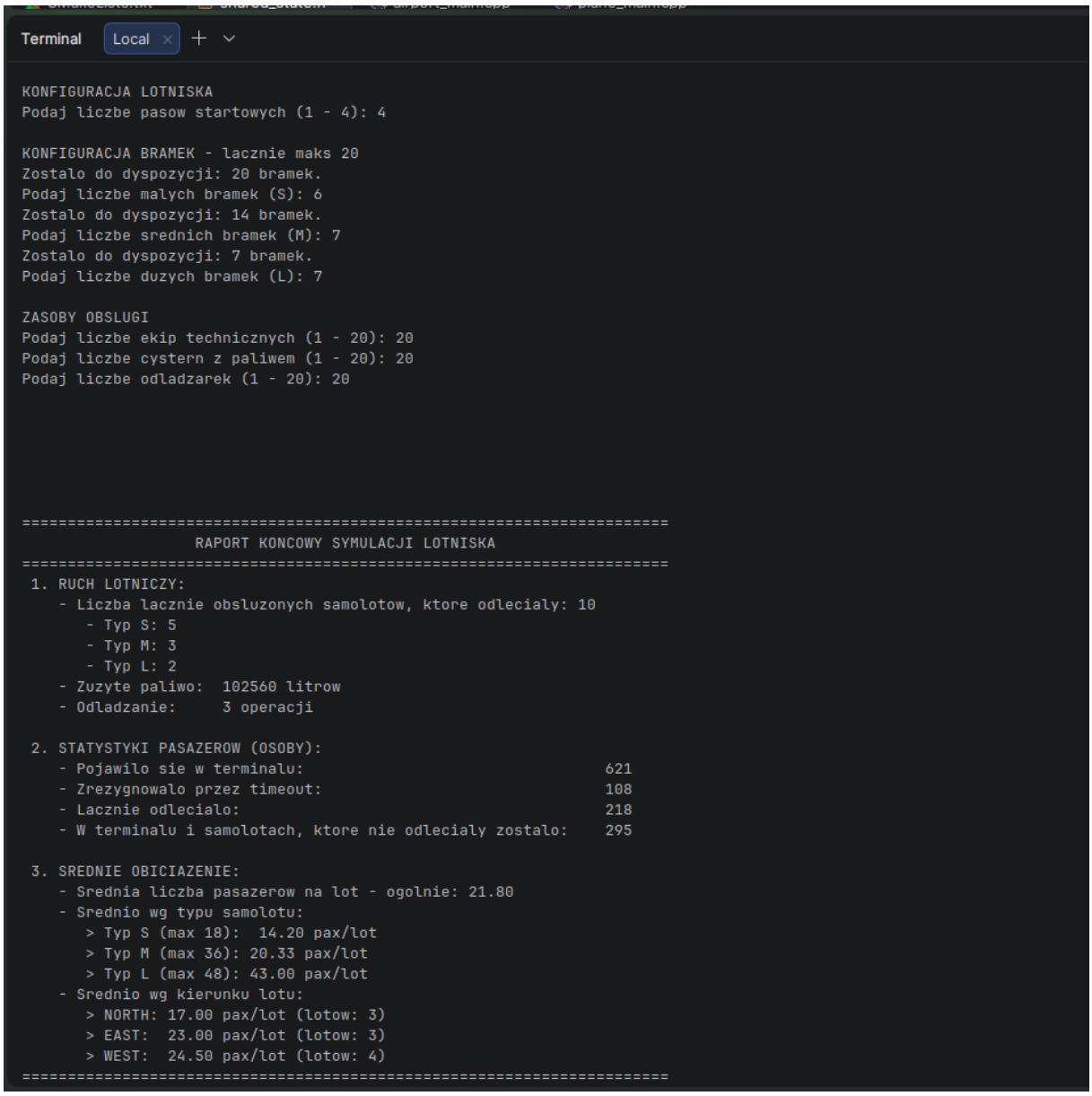


Rys. 5.6: Problem braku paliwa - scenariusz maksymalny



Rys. 5.7: Statystyki - scenariusz maksymalny

Ten sam scenariusz testowy powtórzono dla zbiornika paliwa mającego na start **20000** litrów więcej niż wersja podstawowa. Zmiana ta wyeliminowała problem czekania na paliwo. W ten sposób mniej osób zrezygnowało ze swoich lotów. Rysunek 5.8 przedstawia statystyki końcowe dla wersji z powiększonym zbiornikiem paliwa na start.



Rys. 5.8: Statystyki - scenariusz maksymalny - zwiększona startowa ilość paliwa

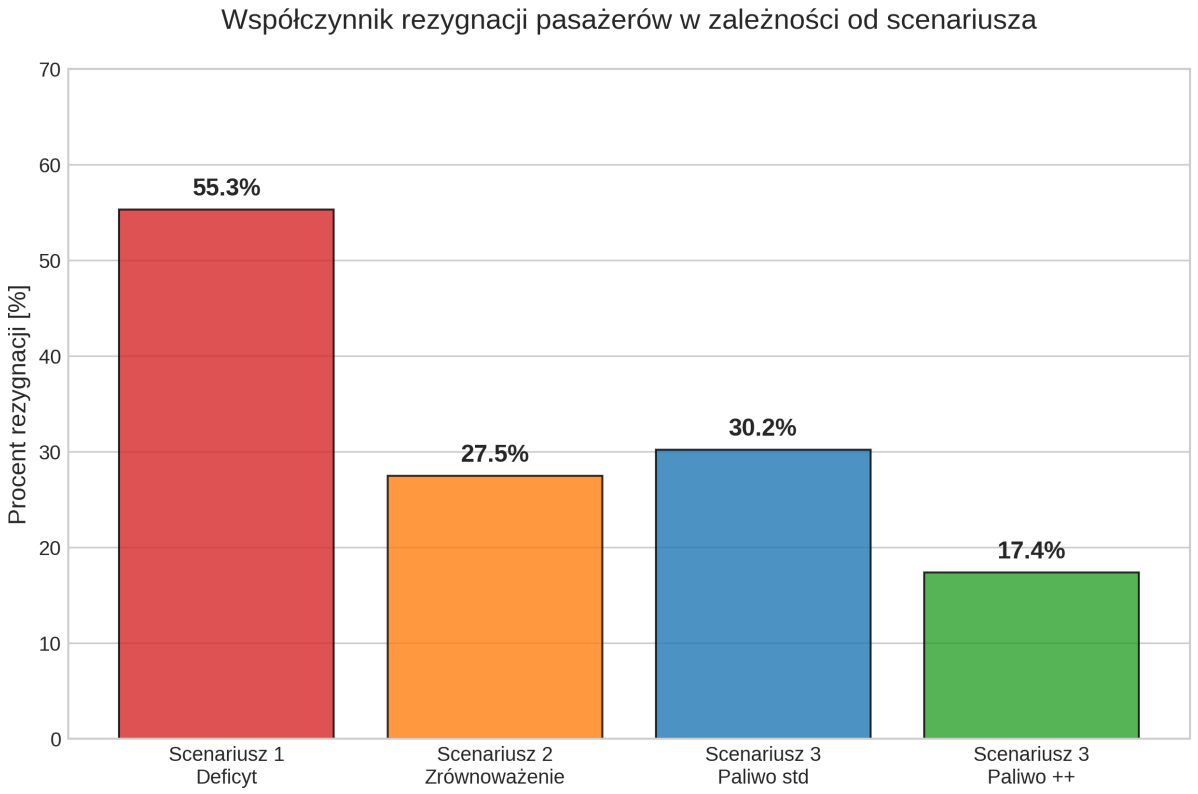
5.5 Podsumowanie i porównanie wyników

W celu ostatecznej weryfikacji wpływu dostępności zasobów na przepustowość lotniska, zestawiono kluczowe statystyki ze wszystkich przeprowadzonych testów. W każdym przypadku obsłużono - start i lądowanie - dokładnie 10 samolotów, jednak różnice w konfiguracji wpłynęły znacząco na komfort pasażerów i płynność operacji. Szczególną uwagę należy zwrócić na współczynnik rezygnacji pasażerów, który jest najlepszym miernikiem opóźnień w systemie.

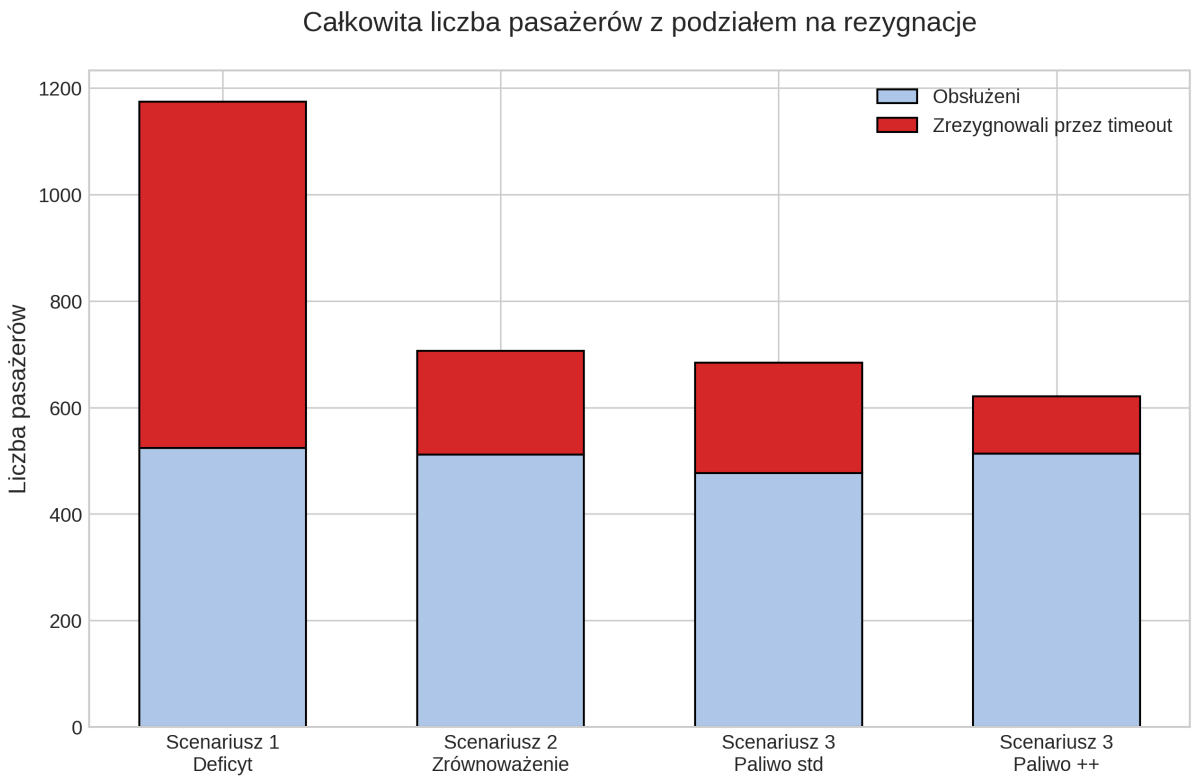
Tab. 5.4: Tabela porównawcza efektywności obsługi w różnych scenariuszach

Parametr	**Scenariusz 1 Deficyt**	**Scenariusz 2 Równowaga**	**Scenariusz 3 Max - paliwo std**	**Scenariusz 3 Max - paliwo ++**
Liczba pasów startowych	1	3	4	4
Liczba samolotów	10	10	10	10
Średnie obłożenie (pax/lot)	29,80	21,90	20,30	21,80
Pasażerowie (w terminalu)	1174	706	684	621
Pasażerowie zrezygnowali	650	194	207	108
Współczynnik rezygnacji	**55,3%**	**27,5%**	**30,2%**	**17,4%**

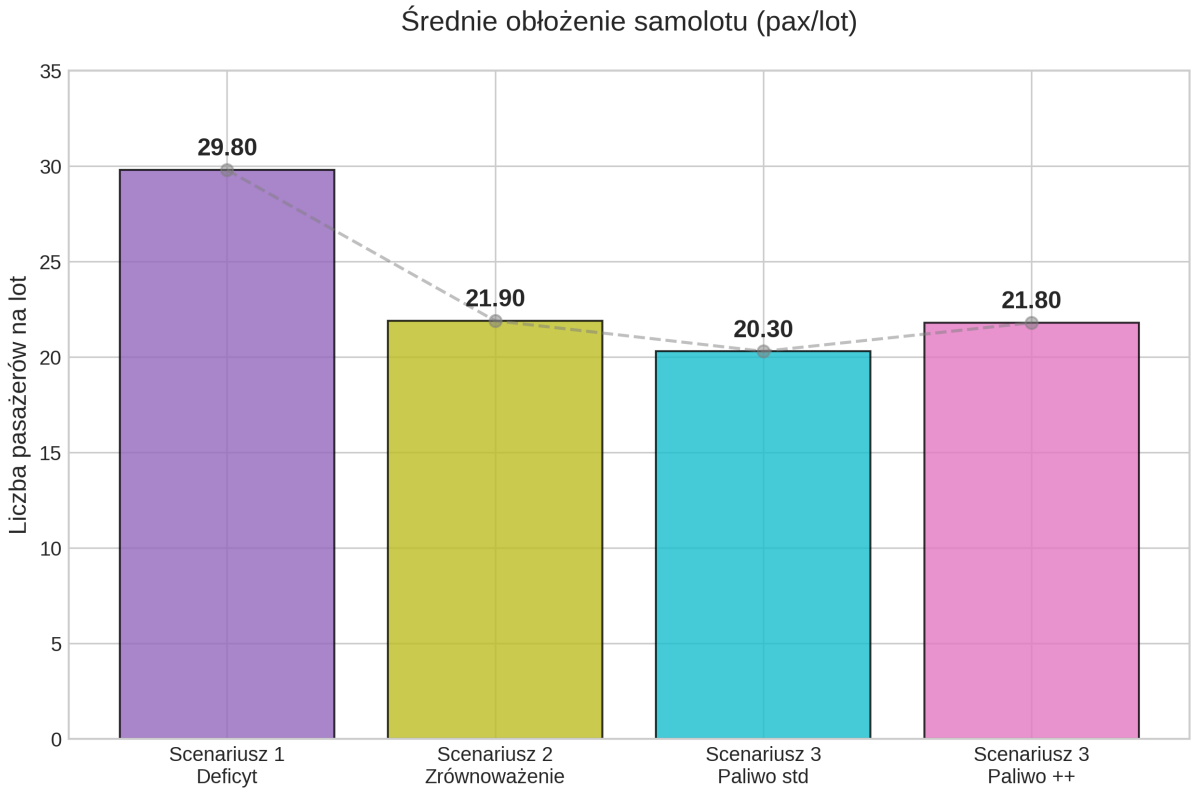
Dla lepszego zobrazowania różnic w wydajności poszczególnych konfiguracji, kluczowe wskaźniki przedstawiono w formie graficznej na Rysunkach 5.9, 5.10 oraz 5.11.



Rys. 5.9: Porównanie współczynnika rezygnacji pasażerów



Rys. 5.10: Struktura pasażerów z podziałem na obsłużonych i rezygnujących



Rys. 5.11: Średnie obłożenie samolotu

Wnioski z analizy danych:

- **Scenariusz 1 -** wysoki współczynnik rezygnacji potwierdza niewydolność systemu przy minimalnych zasobach. Ciekawostką jest najwyższe średnie obłożenie samolotu (29,80 pax/lot). Wynika to z faktu, że samoloty spędzały przy bramkach bardzo dużo czasu oczekując na cysternę lub ekipę, co pozwalało na boarding większej liczby pasażerów przed odlotem, kosztem sporych opóźnień.
- **Paradoks scenariusza 3 w wersji podstawowej i scenariusza 2 -** mimo maksymalnej liczby pasów i sprzętu, współczynnik rezygnacji - 30,2% - był wyższy niż w scenariuszu 2 - 27,5%. Potwierdza to zdiagnozowane wcześniej wąskie gardło związane z brakiem paliwa, a nie samych cystern.
- **Scenariusz 3 - zoptymalizowany -** po zwiększeniu startowego zapasu paliwa, lotnisko osiągnęło najwyższą wydajność. Współczynnik rezygnacji spadł do rekordowo niskiego poziomu 17,4%. Niższe średnie obłożenie lotu - 21,80 - w porównaniu do scenariusza 1 świadczy o płynności – samoloty są obsługiwane sprawnie i odlatują o czasie, nie czekając na dopychanie pasażerów w nieskończoność.

Rozdział 6

Napotkane problemy

Podczas implementacji projektu najbardziej istotnym wyzwaniem było zarządzanie cyklem życia procesów, co początkowo prowadziło do powstawania procesów Zombie, które blokowały zasoby systemowe. Problem ten rozwiązano poprzez dodanie w pętli głównej mechanizmu asynchronicznego odbierania statusów zakończenia. Trudnością była również trwałość pamięci współdzielonej, która po awaryjnym zamknięciu programu uniemożliwiała ponowne uruchomienie symulacji. Wymusiło to implementację obsługi sygnałów gwarantującą zwolnienie zasobów przy wyjściu. Dodatkowo, aby wyeliminować błędy wizualne wynikające z jednoczesnego dostępu do danych wprowadzono semafor synchronizujący warstwę prezentacji, a blokowanie się procesów na zajętych zasobach rozwiązano poprzez zmianę strategii oczekiwania na aktywne sprawdzanie dostępności. Ostatnim, lecz kluczowym problemem zidentyfikowanym podczas testów obciążeniowych - scenariusz 3 - okazała się dostępność zasobu zużywalnego. Mimo skonfigurowania maksymalnej liczby zasobów technicznych, wąskim gardłem stała się ilość paliwa w centralnym zbiorniku. Przy równoczesnej obsłudze wielu samolotów, startowy zapas paliwa wyczerpywał się zbyt szybko w stosunku do cyklu dostaw. Prowadziło to do paradoksalnej sytuacji, w której samolot posiadał przydzieloną cysternę, ale musiał oczekiwać na dostawę surowca - co widać na Rysunku 5.6 w poprzednim rozdziale - co z kolei przekładało się na zwiększoną liczbę rezygnacji pasażerów. Problem ten wyeliminowano poprzez rekalibrację parametrów początkowych symulacji – zwiększenie startowej ilości paliwa o 20 000 litrów pozwoliło na obsłużenie piku zapotrzebowania bez przestojów.

Rozdział 7

Wnioski

Realizacja projektu symulacji portu lotniczego pozwoliła na praktyczną weryfikację wiedzy z zakresu programowania systemowego w środowisku Linux, ze szczególnym uwzględnieniem aspektów współbieżności. Wybór architektury wieloprocesowej okazał się trafny z punktu widzenia modelowania autonomicznych jednostek, zapewniając wysoki poziom izolacji – awaria jednego procesu samolotu nie destabilizowałaby działania całego systemu, co jest istotną przewagą nad modelami wielowątkowymi. Należy jednak pamiętać, że podejście to wiąże się z większym narzutem systemowym wynikającym z kosztownego przełączania kontekstu oraz funkcji `fork`, co przy próbie masowego skalowania mogłoby stanowić ograniczenie wydajnościowe. Zastosowanie pamięci współdzielonej potwierdziło swoją efektywność jako najszybszej metody komunikacji międzyprocesowej IPC, eliminując narzut kopiowania danych, jednak wymusiło rygorystyczne podejście do synchronizacji. Projekt wykazał, że samo użycie binarnych blokad jest niewystarczające. Niezbędne okazało się umiejętne wykorzystanie semaforów licznikowych oraz strategii aktywnego poszukiwania zasobów, co skutecznie zapobiegało zakleszczeniom i zjawisku nieskończonego oczekiwania. Dodatkowo, implementacja problemu producenta i konsumenta przy symulacji tankowania udowodniła, że mechanizm warunkowego usypiania procesów jest znacznie bardziej optymalny dla wykorzystania czasu procesora niż techniki aktywnego odpytywania. Istotne wnioski płyną również z analizy przeprowadzonych scenariuszy testowych. Wykazały one fundamentalną różnicę między dostępnością zasobów technicznych, a dostępnością surowca. Sytuacja zatoru w scenariuszu 3 udowodniła, że w systemach złożonych optymalizacja tylko jednego parametru - zwiększenie liczby cystern - jest nieskuteczna bez zapewnienia odpowiedniej przepustowości warstwy dostaw paliwa. Jest to cenna obserwacja, mająca zastosowanie w projektowaniu wydajnych systemów rozproszonych. Ostatecznie projekt pokazał wagę świadomego zarządzania zasobami systemowymi, w tym konieczność obsługi cyklu życia procesów i eliminacji procesów Zombie. Całość prac potwierdziła, że mechanizmy standardu POSIX, mimo wysokiego stopnia skomplikowania i trudności w debugowaniu, stanowią potężne i elastyczne narzędzie pozwalające na wierne modelowanie złożonych, rzeczywistych systemów rozproszonych.

Bibliografia

1. Tanenbaum, A. S., *Modern Operating Systems*, https://os.ecci.ucr.ac.cr/slides/Andrew-S.-Tanenbaum-Modern-Operating-Systems.pdf
2. Hall, B. J., *Beej's Guide to Unix IPC*, https://beej.us/guide/bgipc/

Spis rysunków

Spis tabel

Spis listingów