Środowisko uruchomieniowe projektu:

- System operacyjny (host) - `Kubuntu 24.04.1 LTS x86_64 (Linux 6.8.0-90-generic)`
- IDE: `CLion 2025.3.1.1`
- Kompilator `C++23 (g++ (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0) + CMake`
- `Zewnętrzne pakiety:`
  - `nlohmann::json v3.11.3` https://github.com/nlohmann/json.git
  - `FTXUI v6.1.9` https://github.com/ArthurSonzogni/FTXUI.git
  - `gtest (nieużywany)`

1. Założenia projektowe

1.1. Podstawowe

- Fabryka produkuje dwa rodzaje czekolady.
- Na stanowisku produkcyjnym 1 jest produkowana czekolada ze składników A, B i C.
- Na stanowisku produkcyjnym 2 jest produkowana czekolada ze składników A, B i D.
- Składniki przechowywane są w magazynie o pojemności N jednostek.
- Składniki A i B zajmują jedną jednostkę magazynową, składnik C dwie, a składnik D trzy jednostki.
- Składniki pobierane są z magazynu, przenoszone na stanowisko produkcyjne 1 lub 2 i używane do produkcji czekolady (typ_1 lub typ_2).
- Jednocześnie trwają dostawy składników A, B, C i D do magazynu.
- Składniki pochodzą z 4 niezależnych źródeł i dostarczane są w nieokreślonych momentach czasowych.
- Fabryka przyjmuje do magazynu maksymalnie dużo składników dla zachowania płynności produkcji.
- Fabryka kończy pracę po otrzymaniu polecenia_1 od dyrektora.
- Magazyn kończy pracę po otrzymaniu polecenia_2 od dyrektora.
- Dostawcy przerywają dostawy po otrzymaniu polecenia_3 od dyrektora.
- Fabryka i magazyn kończą pracę jednocześnie po otrzymaniu polecenia_4 od dyrektora.
- Aktualny stan magazynu zapisany w pliku, po ponownym uruchomieniu stan magazynu jest odtwarzany z pliku.

1.2. Dodatkowe

- Pracownicy działają na podstawie przepisów - przepis określa wejście i wyjście pracownika
- Pracownik 1 i 2 mają ten sam kod
- Dostawcy działają na podstawie wzorców składników - wzorzec określa składnik dostarczany i średni odstęp dostarczania
- Dostawcy A, B, C, D mają ten sam kod
- “Dynamicznie” przydzielana pamięć dla symulacji magazynu (*patrz 4.1*)

2. Opis projektu

2.1. Obiekty

`Item` — reprezentuje pojedynczy typ przedmiotu w magazynie, przechowując jego nazwę, rozmiar i ilość oraz udostępniając operacje porównywania, łączenia/rozłączania stosów i serializacji do JSON.

Istotna poprawka (21.01.2026): Item stał się klasą generyczną (template) z parametrem size - sprawia to że rozmiar przedmiotu to **n*16B**, z czego n to liczba jednostek magazynowych zajmowanych przez przedmiot

`ItemTemplate` — opisuje szablon przedmiotu z nazwą, rozmiarem i bazowym opóźnieniem, umożliwiając generowanie konkretnych obiektów `Item` z losowo modyfikowanym czasem opóźnienia.

`Recipe` — definiuje recepturę produkcji, określając wymagane przedmioty wejściowe i wynikowy produkt oraz sprawdzając, czy z dostępnych zasobów można go wytworzyć.

`SharedVector<T, Capacity>` — implementuje wektor o stałej pojemności przystosowany do współdzielonej pamięci, zapewniając bezpieczny dostęp, usuwanie elementów oraz serializację i deserializację do JSON.

`Message` — reprezentuje komunikat logowania zawierający poziom ważności, treść, identyfikator procesu i znacznik czasu, przeznaczony do przesyłania i czytelnego formatowania logów.

2.2. IPCs

`MessageQueue<T>` — stanowi opakowanie RAII dla kolejek komunikatów System V, umożliwiając bezpieczne tworzenie lub podłączanie do kolejki, wysyłanie i odbieranie komunikatów oraz automatyczne sprzątanie zasobów systemowych.

`Semaphore` — zapewnia obiektowe, bezpieczne w użyciu (RAII) opakowanie dla semaforów System V, umożliwiając synchronizację procesów poprzez blokowanie, odblokowywanie i automatyczne zarządzanie zasobem. Realizuje mutex.

`SharedMemory<T>` — zapewnia RAII-opakowanie dla segmentów pamięci współdzielonej System V, umożliwiając tworzenie, podłączanie, bezpieczny dostęp do danych oraz automatyczne odłączanie i usuwanie segmentu przy właścicielu.

2.3. Stacje

`Warehouse` — reprezentuje magazyn z bezpiecznym dostępem współbieżnym, przechowując przedmioty w pamięci współdzielonej i chroniąc operacje za pomocą semafora działającego jak mutex, z automatycznym tworzeniem i czyszczeniem zasobów IPC.

`WarehouseStats` — przechowuje statystyki magazynu, takie jak nazwa, pojemność, liczba typów przedmiotów, aktualne użycie i lista przechowywanych przedmiotów, umożliwiając łatwe raportowanie i wyświetlanie danych.

2.4. Procesy

`ProcessController` — zarządza cyklem życia procesu w bezpieczny sposób, umożliwiając jego uruchamianie, zatrzymywanie, wstrzymywanie, wznawianie i przeładowanie, przy jednoczesnym udostępnianiu statystyk przez pamięć współdzieloną i obsłudze sygnałów systemowych.

`ProcessStats` — przechowuje bieżące statystyki procesu, w tym jego stan (`CREATED`, `RUNNING`, `PAUSED`, `RELOADING`, `STOPPED`), liczbę wykonanych pętli, liczbę przeładowań i identyfikator procesu.

2.5. Aktorzy

`IRunnable` — interfejs definiujący proces, który można uruchamiać, wstrzymywać, wznawiać, przeładowywać i zatrzymywać, umożliwiając jego kontrolę i monitorowanie przez `ProcessController`.

`Deliverer` — proces realizujący dostawy przedmiotów do magazynu według szablonu `ItemTemplate`, obsługiwany w osobnym wątku i kontrolowany przez `ProcessController` poprzez start, pauzę, wznowienie i przeładowanie, z raportowaniem stanu do `ProcessStats`.

`Worker` — proces przetwarzający receptury, pobierając przedmioty z magazynu wejściowego, wytwarzając produkty zgodnie z `Recipe` i umieszczając je w magazynie wyjściowym, kontrolowany przez `ProcessController` z możliwością startu, pauzy, wznowienia i przeładowania oraz raportowaniem stanu do `ProcessStats`.

LogCollector — proces działający w osobnym wątku, który zbiera i zapisuje komunikaty logów do pliku lub bufora współdzielonego, kontrolowany przez `ProcessController` z możliwością startu, pauzy, wznowienia i przeładowania oraz raportowaniem stanu do `ProcessStats`.

2.6. Logowanie

IQueue<T> — interfejs generycznej kolejki umożliwiający wysyłanie i odbieranie wiadomości dowolnego typu `T`, do implementacji jako kolejka systemowa, pamięciowa lub mock.

Logger — uniwersalny logger z różnymi poziomami (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `FATAL`), obsługujący formatowanie printf-style i opcjonalną wysyłkę komunikatów do kolejki (`IQueue<Message>`).

MockQueue<T>: dziedziczy po `IQueue<T>` i służy do testów — `send(T m)` wypisuje wiadomość na `std::cout`, `receive(T* m)` nie robi nic.

2.7. Usługi

DelivererService - zarządza cyklem życia wszystkich dostawców, tworząc, usuwając, pauzując, wznawiając i przeładowując ich instancje za pomocą `ProcessController`, przy jednoczesnym utrzymaniu bezpiecznych statystyk w pamięci współdzielonej.

LoggerService - zarządza procesem `LogCollector` i kolejką komunikatów, zapewniając RAII-style obsługę logowania z dostępem do współdzielonego bufora wiadomości i root loggera.

WarehouseService - zarządza tworzeniem, dostępem i usuwaniem instancji magazynów oraz udostępnia ich statystyki i listę wszystkich magazynów w systemie.

WorkerService - zarządza tworzeniem, dostępem, wstrzymywaniem, wznawianiem i usuwaniem instancji pracowników oraz udostępnia ich statystyki i listę wszystkich pracowników w systemie.

2.8. Dyrektor

Supervisor - zarządza zakończeniem działania usług (`WarehouseService`, `WorkerService`, `DelivererService`) przy użyciu wątko-bezpiecznego tokena `ShutdownToken`; zapewnia metody `stop_deliverers()`, `stop_workers()`, `stop_warehouses()` i `stop_all()`.

2.9. GUI

ControlPanel - udostępnia pionowy zestaw przycisków FTXUI, które pozwalają użytkownikowi zatrzymać pracowników, dostawców, magazyny lub całą symulację, wywołując odpowiednie metody powiązanego obiektu `Supervisor`.

Dashboard - tworzy interfejs zakładkowy, umożliwiający użytkownikowi przełączanie się między widokami magazynów, pracowników i dostawców, wyświetlając aktualnie wybraną zakładkę w centralnym komponencie FTXUI.

LogPanel - tworzy panel w FTXUI, który pobiera logi z `LoggerService`, koloruje je według poziomu ważności i automatycznie przewija do najnowszych wpisów.

Layout - komponuje główny interfejs aplikacji w FTXUI, układając dashboard i panel sterowania w górnym wierszu oraz panel logów w dolnej, elastycznej sekcji, tworząc spójną, renderowalną strukturę UI.

WarehouseTable - renderuje tabelę z danymi magazynów pobranymi z `WarehouseService`.

DelivererTable - renderuje tabelę z danymi dostawców pobranymi z `DelivererService`.

`WorkerTable` - renderuje przewijaną tabelę z danymi pracowników pobranymi z `WorkerService`.

2.10. Main

Główna pętla programu tworzy i inicjalizuje wszystkie kluczowe komponenty symulacji fabryki: loger, usługi magazynowe, pracownicze i dostawcze, a następnie konfiguruje magazyny, dostawców i pracowników wraz z przypisanymi im recepturami i szablonami produktów. Po skonfigurowaniu komponentów, pętla uruchamia interfejs terminalowy przy użyciu FTXUI, tworząc układ zawierający panel sterowania, panel logów oraz tabele stanów magazynów, pracowników i dostawców w formie dashboardu. W osobnym wątku działa „refresher", który co 20 milisekund wysyła zdarzenie do ekranu, odświeżając UI i reagując na sygnał zakończenia (`ShutdownToken`). Pętla główna pozostaje aktywna aż do momentu, gdy token sygnalizuje zakończenie pracy, po czym wątek odświeżający jest dołączany, a program kończy działanie.

3. Funkcjonalności - co udało się zrobić

- Implementacja wymaganych poleceń - są dostępne jako przyciski w UI; polecenia pozwalają na sekwencyjne uruchamianie, tj. możemy zakończyć pracę dostawców, a następnie niezależnie zakończyć pracę magazynu
- Implementacja zapisu i odczytu do pliku - obrany format to JSON (najlepiej odzwierciedla model składania stosów przedmiotów)
- Implementacja raportowania (logów) zrzucanych do pliku - podczas trwania symulacji generowany jest plik z logami, które zostały wyłane przez procesy składowe symulacji do kolejki komunikatów
- Dostarczanie składników w nieokreślonych momentach czasowych zostało zaimplementowane jako losowy czas dostarczenia składnika

4. Problemy implementacyjne

4.1. Umieszczenie dynamicznej tablicy std::vector w pamięci dzielonej

`std::vector<T>` nie nadaje się do pamięci dzielonej, bo trzyma wskaźnik do sterty, który nie działa między procesami. Dlatego potrzebna jest osobna struktura `SharedVector<T, Capacity>`, która przechowuje **rozmiar i wszystkie elementy w jednym bloku** w shared memory. `Capacity` jako parametr typu pozwala mieć **stały, znany w czasie kompilacji rozmiar**, eliminując dynamiczną alokację i zapewniając przewidywalny layout między procesami.

4.2. Logowanie dla obiektu kolejki komunikatów

Jeśli kolejka (`MessageQueue` lub `IQueue`) podczas `send()` lub `receive()` używa loggera, który sam wysyła komunikaty do tej samej kolejki, powstaje **nieskończona rekurencja**: kolejka wywołuje logger, logger próbuje wysłać komunikat do kolejki, która znów wywołuje logger itd. Dlatego implementacja kolejki w IPC musi być **bezpieczna dla logowania**: albo nie loguje wcale, albo loguje wyłącznie lokalnie (np. do `stdout` lub `stderr`) bez korzystania z tej samej kolejki. Dlatego, nawet jeśli udało się zaimplementować logowanie widoczne na UI - część logów musi zostać przesłana do `stdout` lub `stderr`

5. Elementy specjalne

5.1. Interfejs graficzny

Interfejs stworzony przy użyciu `ControlPanel`, `Dashboard`, `LogPanel` i `Layout` wraz z tabelami (`WarehouseTable`, `DelivererTable`, `WorkerTable`) daje użytkownikowi możliwość:

- Zarządzania symulacją poprzez wydawanie poleceń w `ControlPanel`.
- Monitorowania logów w `LogPanel`, z kolorowaniem według poziomu ważności i automatycznym przewijaniem do najnowszych wpisów.
- Przeglądania danych magazynów, dostawców i pracowników w tabelach `WarehouseTable`, `DelivererTable` i `WorkerTable`, które wyświetlają aktualne wartości statystyk.
- Obserwowania zmian w czasie rzeczywistym dzięki odświeżaniu danych w pętli FTXUI.

5.2. Wielopoziomowy system logowania z kolejką komunikatów

W systemie zastosowano wielopoziomowy mechanizm logowania, składający się z `Logger`, `MessageQueue` i `LogCollector`. `Logger` generuje komunikaty, które mogą być wysyłane do `MessageQueue` – abstrakcyjnej kolejki komunikatów umożliwiającej przekazywanie danych między procesami. `LogCollector` działa jako osobny proces, odbierający wiadomości z kolejki i zapisujący je do wspólnej pamięci (`SharedMemory`) oraz plików. Dzięki temu oddzielenie kolektora od głównej symulacji zapewnia, że logowanie nie blokuje działania procesu głównego i umożliwia współdzielenie danych pomiędzy procesami. Użycia osobnego procesu wymusza zastosowanie pamięci dzielonej, ponieważ klasyczne zmienne w pamięci lokalnej nie byłyby dostępne dla `LogCollector`.

5.3. Scentralizowane zarządzanie procesami za pomocą sygnałów

System wykorzystuje scentralizowane zarządzanie procesami poprzez klasy `ProcessController` i `IRunnable`. Każdy proces implementuje interfejs `IRunnable`, który definiuje standardowe metody cyklu życia (`run`, `stop`, `pause`, `resume`, `reload`) oraz udostępnia statystyki w `ProcessStats`. `ProcessController` pełni rolę menedżera, wysyłając sygnały systemowe (np. `SIGTERM`, `SIGSTOP`, `SIGCONT`, `SIGHUP`) do kontrolowanych procesów w celu zatrzymania, wstrzymania, wznowienia lub przeładowania ich stanu.

5.4. Klasy obsługujące SystemV IPC - RAII

W systemie klasy procesów i usług (oraz opakowania IPC `MessageQueue, Semaphore, SharedMemory`) stosują mechanizm **RAII** – Resource Acquisition Is Initialization – co oznacza, że alokacja zasobów (procesów, kolejek IPC, pamięci dzielonej) odbywa się w konstruktorze, a zwolnienie i bezpieczne zakończenie procesów następuje w destruktorze.

5.5. Generalizacja przedmiotów/przepisów i opis składników/produktów

W systemie symulacji produkcji **przedmioty** i **przepisy** zostały uogólnione jako obiekty `Item` i `Recipe`. Każdy `Item` posiada nazwę, ilość i jednostkę produkcji/dostarczania, co pozwala traktować zarówno surowce, jak i produkty końcowe w jednolity sposób. `Recipe` definiuje zestaw wymaganych `Item`ów jako składniki oraz jeden lub więcej `Item`ów jako produkty, umożliwiając elastyczne opisanie procesu przetwarzania w systemie. Dzięki takiej generalizacji system łatwo obsługuje nowe kombinacje składników i produktów, a pracownicy (`Worker`) i dostawcy (`Deliverer`) mogą korzystać z tej samej abstrakcji do realizacji transportu i produkcji.

6. Funkcje systemowe - linki

6.1. System plików

- LogCollector::_open_file - otwieranie pliku logowania - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/actors/LogCollector.cpp#L109-L121
- LogCollector::_main - wpisywanie do pliku (strumień) - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/actors/LogCollector.cpp#L76C1-L94C2
- Warehouse::_read_file - otwieranie pliku i czytanie z pliku (strumień) - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/stations/Warehouse.cpp#L206C1-L217C2
- Warehouse::_write_file - otwieranie pliku i wpisywanie do pliku (strumień) - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/stations/Warehouse.cpp#L185-L204
- make_key - tworzenie katalogów, otwieranie i tworzenie pliku - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/ipcs/key.h#L29-L46

6.2. Tworzenie procesów

- ProcessController::run - tworzenie procesu potomnego fork + exec - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/20ba74a5ab2221d83a7cbf17172b1233438249c5/include/processes/ProcessController.cpp#L46-L62
- ProcessController::~ProcessController - oczekiwanie na zakończenie procesu potomnego https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/processes/ProcessController.cpp#L34-L45

6.3. Tworzenie i obsługa wątków

- main - tworzenie wątku z f. anonimową, łączenie wątków - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/src/main.cpp#L110C1-L116C8

6.4. Obsługa sygnałów

- ProcessController::stop, pause, resume, reload - wysyłanie sygnału - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/processes/ProcessController.cpp#L65-L88
- ProcessController::_handle_stop, _handle_pause, _handle_reload, _handle_resume - obsługa sygnałów - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/processes/ProcessController.cpp#L99-L114

6.5. Ftok

- make_key - tworzenie klucza IPC - https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/ipcs/key.h#L29-L46

6.6. Semafory

Cykl życia semafora jest obsłużony w klasie Semaphore:

- Semaphore -> semget, semctl
- ~Semaphore -> semctl
- lock -> semop
- unlock -> semop
- value -> semctl

https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/ipcs/Semaphore.cpp#L12-L84

6.7. Pamięć dzielona

Cykl życia pamięci dzielonej jest obsłużony w klasie SharedMemory:

- SharedMemory<T> -> shmget, shmat
- ~SharedMemory<T> -> shmdt, shmctl

https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/ipcs/SharedMemory.h#L135-L223

6.8. Kolejka komunikatów

Cykl życia kolejki jest obsłużony w klasie MessageQueue:

- MessageQueue<T> -> msgget
- ~MessageQueue<T> -> msgctl
- send -> msgsnd
- receive -> msgrcv

https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/a9d182fbb8860f3ffd04f164da52409af5a74a9e/include/ipcs/MessageQueue.h#L104-L191

7. Testy

7.1. Test podstawowy - *Deliverer (tests/Deliverer.cpp)*

Test ma za zadanie sprawdzić:
- Czy proces dostawcy dostarcza przedmiot
- Czy proces dostawcy poprawnie reaguje na sygnały
- Czy magazyn jest w stanie przyjąć przedmiot
- Czy zasoby są likwidowane poprawnie

Wynik:
- Dostawca dostarcza przedmiot dokładnie raz -> jest ograniczony sygnałami SIGUSR1 i SIGCONT
- Magazyn ma 1 przedmiot
- Dostawca (i zbieracz logów) “sprzątają po sobie” - odłączają pamięć dzieloną, a dyrektor usuwa IPCs

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t1_2026_01_22_00_47_10.log

7.2. Test podstawowy - *Worker (tests/Worker.cpp)*

Test ma za zadanie sprawdzić:
- Czy proces pracownika pobiera, a następnie produkuje przedmiot
- Czy proces pracownika reaguje na sygnały
- Czy magazyn jest w stanie wydać przedmiot
- Czy zasoby zą likwidowane poprawnie

Wynik:
- Pracownik pobiera przedmiot w kroku 1 i produkuje w kroku 2
- Pracownik (i zbieracz logów) “sprzątają po sobie” - odłączają pamięć dzieloną, a dyrektor usuwa IPCs

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t2_2026_01_22_00_47_42.log

7.3. Test zrównoleglenia - *Deliverer (StopDeliverers.cpp)*

Test ma za zadanie sprawdzić:
- Czy przy dużej ilości procesów (64) nie dochodzi do wyścigu i blokady
- Czy proces dyrektora jest w stanie centralnie zakończyć procesy za pomocą zestawu sygnałów (polecenie StopDeliverers)

Wynik:
- Magazyn zapełniany jest poprawnie, do końca
- Polecenie jest wykonane do końca - ale nie od razu, konieczne jest znaczne oczekiwanie…

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t3_redacted.log
[log został zredagowany z ostrzeżeń dotyczących przepełnienia magazynu]

7.4. Test zrównoleglenia - *Worker (StopWorkers.cpp)*

Test ma za zadanie sprawdzić:
- Czy przy dużej ilości procesów (64) nie dochodzi do wyścigu i blokady
- Czy proces dyrektora jest w stanie centralnie zakończyć procesy za pomocą zestawu sygnałów (polecenie StopDeliverers)

Wynik:
- Zawartość magazynu jest poprawnie transferowana z T4A do T4B (cała)
- Polecenie jest wykonane do końca, sprawnie i

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t4_redacted.log
[log został zredagowany z ostrzeżeń dotyczących przepełnienia magazynu i pobrania z pustego magazynu]

7.5. Test odświeżenia *(StopWarehouses.cpp)*

Test ma za zadanie sprawdzić:
- Czy procesy poprawnie wykrywają brak magazynu przy SIGHUP

Wynik:
- Procesy po usunięciu magazynu mają status PAUSED
- Procesy poprawnie wykryły brak magazynu - nie próbują się do niego dobić

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t5_2026_01_22_00_53_52.log

7.6. Test odświeżenia łączonego *(StopChained.cpp)*

Test ma za zadanie sprawdzić:
- Czy supervisor poprawnie pomija zatrzymane procesy przy odświeżaniu

Wynik:
- Zatrzymane procesy nie otrzymały sygnału
- Pozostałe procesy mają status PAUSED

Raport (log):
https://github.com/mateusz-gluch-pk/so-projekt-fabryka-czekolady/blob/main/test-reports/t6_redacted.log