Raport z Projektu: Systemy Operacyjne

Temat 3 – Fabryka Czekolady (Problem Producenta i Konsumenta)

Imię i Nazwisko	Nr albumu
OLIWIA SKAŁKA	**155223**

0. Wstęp i Instrukcja Uruchomienia

Opis Projektu

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 podzespołów 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.

Napisz programy dla procesów dyrektor, dostawca i pracownik reprezentujących odpowiednio: dyrektora, dostawców produktów A, B, C i D oraz pracowników na stanowiskach 1 i 2. Raport z przebiegu symulacji zapisać w pliku (plikach) tekstowym.

Wymagania Systemowe

Projekt został przygotowany dla środowiska systemu **Linux** (lub kompatybilnego, np. WSL). Do poprawnego działania wymagane są:

- Kompilator **g++** obsługujący standard C++ (min. C++11).
- Narzędzie **make** do automatyzacji procesu kompilacji.
- Biblioteki systemowe do obsługi IPC System V (`sys/shm.h`, `sys/sem.h`).

Kompilacja (Makefile)

Projekt wykorzystuje plik `Makefile` do zarządzania kompilacją trzech niezależnych plików wykonywalnych.

Komendy zdefiniowane w Makefile:

- `make all` (lub po prostu `make`) – Kompiluje pliki źródłowe i tworzy pliki wykonywalne: `dyrektor`, `dostawca`, `pracownik`.

    - *Flagi kompilatora:* `-Wall -std=c++11 -Iinclude` (ostrzeżenia, standard, ścieżka do nagłówków).
- `make clean` – Usuwa pliki binarne, pliki obiektowe, plik stanu magazynu (`.bin`) oraz logi (`.txt`), przywracając katalog do stanu czystego.

Aby wyczyścić projekt ze starych plików (pliki binarne, logi, plik stanu), wpisz w terminalu:

```
make clean
```

Aby skompilować projekt, wpisz w terminalu:

```
make
```

Aby uruchomić projekt, wpisz w terminalu:

```
./dyrektor
```

1. Ogólny opis i założenia projektowe

Celem projektu było stworzenie symulacji fabryki czekolady działającej w środowisku wieloprocesowym systemu Linux. Projekt rozwiązuje klasyczny problem synchronizacji **Producenta i Konsumenta** przy użyciu mechanizmów IPC System V.

Kod został napisany w języku C++. Użyte środowisko: Linux (serwer Torus)

Architektura Systemu

Zgodnie z wymaganiami, zrezygnowano z rozwiązań scentralizowanych na rzecz autonomicznych procesów komunikujących się przez **Pamięć Dzieloną**.

Szczegółowy opis modułów systemu

System został zaprojektowany w architekturze wieloprocesowej, gdzie poszczególne moduły pełnią ściśle określone funkcje:

1. **Dyrektor (`dyrektor.cpp`) – Inicjator i Zarządca Systemu**
    - **Rola:** Proces nadrzędny (Parent), odpowiedzialny za przygotowanie środowiska pracy oraz koordynację cyklu życia całej symulacji.
    - **Inicjalizacja IPC:** Na początku działania tworzy segment pamięci dzielonej (`shmget`) oraz zestaw semaforów (`semget`). Inicjuje wartości semaforów oraz strukturę bufora cyklicznego w pamięci (lub odtwarza ją z pliku `stan_magazynu.bin`, jeśli istnieje).
    - **Tworzenie Procesów:** Wykorzystuje funkcję `fork()` do utworzenia procesów potomnych, a następnie `execl()` do uruchomienia odpowiednich programów wykonawczych:
        - Tworzy **1 proces** `dostawca` (pełniący funkcję kierownika logistyki).

        - Tworzy **2 procesy** `pracownik`, przekazując im w argumencie typ stanowiska ("1" lub "2").
    - **Sterowanie:** Obsługuje interfejs użytkownika (menu w terminalu) oraz przechwytuje sygnały systemowe (`SIGINT`, `SIGUSR1`), zapewniając bezpieczne zamknięcie systemu i zwolnienie zasobów (`IPC_RMID`).

2. **Dostawca (`dostawca.cpp`) – Kierownik Dostaw i Podwykonawcy**
    - **Hierarchia:** Proces ten działa jako zarządca grupy producentów. Wewnątrz swojej funkcji `main()` wykonuje pętlę, w której 4-krotnie wywołuje `fork()`.
    - **Podprocesy:** W ten sposób powołuje do życia **4 niezależne procesy potomne**, z których każdy specjalizuje się w dostarczaniu jednego typu surowca (A, B, C lub D).
    - **Logika Producenta:** Każdy z 4 podprocesów realizuje logikę:
        1. Oczekiwanie na miejsce w magazynie (operacja `P` na semaforze `EMPTY`).
        2. Zajęcie sekcji krytycznej (semafor `MUTEX`).
        3. Wstawienie surowca do bufora cyklicznego (aktualizacja wskaźnika `head`).
        4. Powiadomienie o dostępności towaru (operacja `V` na semaforze `FULL`).
    - Główny proces `dostawca` czeka na zakończenie swoich dzieci (`wait`), zapobiegając powstawaniu procesów osieroconych.

3. **Pracownik (`pracownik.cpp`) – Konsumenci Zasobów**
    - **Rola:** Procesy te symulują pracę linii produkcyjnych. Są uruchamiane bezpośrednio przez Dyrektora i działają niezależnie.
    - **Logika Konsumenta:** Każdy pracownik w nieskończonej pętli próbuje skompletować wymagany zestaw składników (dla Typu 1: A+B+C, dla Typu 2: A+B+D).
    - **Zapobieganie Zakleszczeniom:** Aby uniknąć deadlocka, zaimplementowano tu ścisłą kolejność pobierania zasobów. Pracownik zawsze blokuje semafory `FULL` w kolejności alfabetycznej (np. najpierw A, potem B, na końcu C/D).
    - **Produkcja:** Po pobraniu składników z bufora cyklicznego (aktualizacja wskaźnika `tail`), proces symuluje czas produkcji (`sleep`) i loguje wykonanie zadania, po czym zwalnia miejsce w magazynie (operacja `V` na semaforach `EMPTY`).

4. **Weryfikacja i Raportowanie (`pokaz_stan.cpp`)**
    - **Podgląd Stanu (`./pokaz_stan`):** Zaimplementowano dodatkowe narzędzie, które odczytuje binarny zrzut pamięci (`stan_magazynu.bin`). Pozwala ono w przejrzysty sposób wyświetlić dokładną ilość surowców (A, B, C, D), co umożliwia weryfikację poprawności bilansu po zakończeniu symulacji.
    - **Logowanie Historii:** Wszystkie operacje są na bieżąco zapisywane w pliku tekstowym `raport_symulacji.txt`. Plik ten zawiera pełną historię zdarzeń oraz **końcowy stan magazynu**, a jego analiza pozwala potwierdzić zgodność stanu końcowego z wykonanymi operacjami.

Mechanizm Synchronizacji (10 Semaforów)

Aby uniknąć aktywnego oczekiwania (busy waiting) i wyścigów (race conditions), zastosowano zestaw 10 semaforów:

1. **MUTEX (Indeks 0):** Semafor binarny chroniący sekcję krytyczną (dostęp do struktur danych w pamięci dzielonej).
2. **Pary EMPTY/FULL (Indeksy 1-8):** Dla każdego z 4 surowców (A, B, C, D) stworzono parę semaforów zliczających:
   - `EMPTY_X`: Sygnalizuje ilość wolnego miejsca. Dostawca zasypia na tym semaforze, gdy magazyn jest pełny.
   - `FULL_X`: Sygnalizuje ilość dostępnego towaru. Pracownik zasypia na tym semaforze, gdy brakuje składników.
3. **LOG (Indeks 9):** Semafor binarny synchronizujący dostęp do wyjścia standardowego (`std::cout`) oraz pliku logu, zapobiegający nakładaniu się komunikatów.

Zapobieganie Zakleszczeniom (Deadlock)

Zaimplementowano mechanizm zapobiegania zakleszczeniom poprzez ustalenie ścisłej kolejności pobierania zasobów przez Pracowników. Każdy pracownik zawsze pobiera semafory `FULL` w kolejności alfabetycznej (A -> B -> C/D), co uniemożliwia powstanie cyklu oczekiwania.

---

2. Co udało się zrobić / Elementy specjalne

Projekt spełnia wszystkie wymagania podstawowe:

- **Brak aktywnego oczekiwania:** Wykorzystanie semaforów sprawia, że procesy są usypiane przez system operacyjny, gdy nie mogą wykonać pracy.
- **Obsługa Sygnałów:**
  - `SIGINT` (Ctrl+C): Powoduje bezpieczne zatrzymanie systemu, zapisanie stanu magazynu i zwolnienie zasobów IPC.
  - `SIGUSR1`: Niestandardowy sygnał wywołujący "Niezapowiedzianą kontrolę" – wypisuje aktualny stan magazynu na ekran bez przerywania symulacji. Aby go wywołać, należy w osobnej konsoli wpisać:

    ```
    kill -USR1 [PID_DYREKTORA]
    ```

    (PID jest wyświetlany w pierwszej linii logów po uruchomieniu Dyrektora).
- **Persystencja Danych:** Stan magazynu jest zapisywany do pliku binarnego (`stan_magazynu.bin`) przy zamknięciu i automatycznie odtwarzany przy ponownym uruchomieniu dyrektora.
- **Logowanie Hybrydowe:** Funkcja `loguj_komunikat` zapisuje dane jednocześnie na ekran (z kolorami) i do pliku tekstowego (czysty tekst).

Oraz zawiera dodatkowe funkcjonalności:

- **Kolorowanie składni (ANSI Colors):** W celu zwiększenia czytelności logów w terminalu, wprowadzono rozróżnienie kolorystyczne procesów (Dyrektor: Cyjan, Dostawca: Zielony, Pracownik: Żółty, System: Magenta).

---

3. Problemy i Testy

Napotkane problemy i rozwiązania

1. **Niespójność stanów magazynowych (Przepełnienie / Ujemne stany):**
   - *Problem:* Podczas testów wydajnościowych (po usunięciu funkcji `sleep`), procesy zaczęły działać z maksymalną szybkością. Doprowadziło to do sytuacji wyścigu, w której logiczne warunki `if` nie nadążały, skutkując stanami typu `162/100` lub wartościami ujemnymi.
   - *Rozwiązanie:* Całkowite zastąpienie warunków logicznych w kodzie (`if (ilosc < max)`) mechanizmem **semaforów zliczających** (`EMPTY` i `FULL`). Operacje na semaforach są atomowe na poziomie jądra systemu, co fizycznie uniemożliwia przekroczenie limitów.
2. **Zakleszczenie w obsłudze sygnału (Deadlock przy Ctrl+C):**
   - *Problem:* Program zawieszał się (blokował terminal) po wciśnięciu Ctrl+C. Analiza wykazała, że przyczyną było wywołanie funkcji `loguj_komunikat` wewnątrz handlera `handle_sigint`. Funkcja ta próbowała zająć semafor `SEM_LOG`. Jeśli sygnał nadszedł w momencie, gdy program główny już trzymał ten semafor, następowało zakleszczenie (handler czekał na zasób trzymany przez przerwany proces).
   - *Rozwiązanie:* Usunięto wywołanie funkcji logującej (używającej semaforów) z procedury obsługi sygnału `SIGINT`, pozostawiając jedynie bezpieczne operacje zapisu stanu i zakończenia procesów (`sprzatanie_i_wyjscie`).
3. **Niewystarczająca struktura danych (Int vs Ring Buffer):**
   - *Problem:* W początkowej fazie projektu magazyn był reprezentowany jedynie przez zmienną typu `int` (licznik sztuk). Rozwiązanie to okazało się niewystarczające, ponieważ nie pozwalało na identyfikację konkretnych jednostek towaru ani na realizację kolejki FIFO (First In, First Out), co jest kluczowe dla poprawnej symulacji przepływu materiałów.
   - *Rozwiązanie:* Przebudowano strukturę pamięci dzielonej, zastępując proste liczniki **buforem cyklicznym (Ring Buffer)** dla każdego surowca. Zaimplementowano wskaźniki `head` (dla dostawców) i `tail` (dla pracowników), co pozwala na fizyczne przechowywanie danych w tablicy i ich pobieranie w tej samej kolejności, w jakiej zostały wyprodukowane.
4. **Niespójność danych między raportem a plikiem stanu (Race Condition przy zamykaniu):**
   - *Problem:* Podczas weryfikacji końcowej zauważono rozbieżność w bilansie: logi tekstowe (`raport_symulacji.txt`) wskazywały np. na dostarczenie 1768 sztuk towaru, podczas gdy odczyt pliku binarnego (`stan_magazynu.bin`) po restarcie pokazywał stan odpowiadający 1767 sztukom. Przyczyną była niepoprawna kolejność operacji w procedurze kończenia pracy (Polecenie 4). Dyrektor wykonywał zrzut pamięci do pliku *zanim* upewnił się, że procesy potomne zakończyły działanie. W rezultacie, w ułamku sekundy między zapisem pliku a zabiciem procesów, Dostawca zdążył jeszcze zmodyfikować pamięć i wypisać log, co nie zostało utrwalone w pliku.
   - *Rozwiązanie:* Zmieniono logikę zamykania systemu. Wprowadzono mechanizm "wyciszania": najpierw Dyrektor wysyła sygnały zakończenia i czeka na potwierdzenie śmierci wszystkich procesów potomnych za pomocą funkcji `waitpid()`. Dopiero gdy w systemie nie ma aktywnych procesów modyfikujących pamięć, następuje bezpieczny zapis stanu (`zapisz_stan`) i zwolnienie zasobów. Gwarantuje to idealną zgodność logów z zapisanym stanem.

Scenariusze Testowe i Weryfikacja Działania

Przeprowadzono serię 4 testów, mających na celu weryfikację stabilności, synchronizacji oraz poprawności implementacji mechanizmów IPC.

Test 1: Weryfikacja problemu Producenta i Konsumenta (Przepływ danych)

- **Cel:** Sprawdzenie, czy mechanizm bufora cyklicznego poprawnie obsługuje wstawianie i pobieranie danych oraz czy semafory zapobiegają pobraniu surowców z pustego magazynu (stanów ujemnych).
- **Przebieg:**
  1. Uruchomiono program `./dyrektor`.
  2. Obserwowano logi systemowe przez 30 sekund.
  3. Zwrócono uwagę na sekwencję komunikatów: Dostawca [A] -> Wzrost stanu A -> Pracownik [1] -> Spadek stanu A.
- **Wynik:** System zachował spójność danych. Pracownicy nigdy nie zgłosili wyprodukowania czekolady, jeśli w magazynie brakowało choćby jednego składnika (czekali w stanie uśpienia na semaforze `FULL`). Nie odnotowano stanów ujemnych ani przekłamania liczników.

Test 2: Test semaforów zaporowych (Zapełnienie magazynu)

- **Cel:** Weryfikacja, czy procesy dostawców (producentów) zostaną poprawnie zablokowane przez system operacyjny w momencie przepełnienia bufora (100 sztuk).
- **Przebieg:**
  1. Uruchomiono program i wybrano z menu **Opcję 1 (Stop produkcji)**. Pracownicy przestali pobierać towar.
  2. Dostawcy kontynuowali pracę do momentu zapełnienia magazynów.
  3. Wysłano sygnał `SIGUSR1` (Kontrola), aby podejrzeć stan pamięci dzielonej.
- **Wynik:**
  - Logi dostawców przestały się pojawiać (procesy weszły w stan oczekiwania na operacji `semop`).
  - Raport kontrolny `SIGUSR1` wykazał stan `100/100` dla wszystkich surowców.
  - Żaden surowiec nie przekroczył wartości 100, co dowodzi, że semafor `EMPTY` skutecznie zablokował nadmiarowe operacje zapisu.

Test 3: Weryfikacja trwałości danych (Persistence & Recovery)

- **Cel:** Sprawdzenie, czy system potrafi odtworzyć stan bufora cyklicznego i semaforów po restarcie (wymóg persystencji).
- **Przebieg:**
  1. Doprowadzono do stanu, w którym magazyn był częściowo zapełniony (np. A: 50, B: 30).
  2. Wymuszono zamknięcie programu przez `Ctrl+C` (co wywołało zapisz struktur do pliku binarnego).
  3. Ponownie uruchomiono program `./dyrektor`.
- **Wynik:** Program wykrył plik `stan_magazynu.bin` i wyświetlił komunikat: *"[DYREKTOR] Wznowiono pracę z pliku"*. Odczytane wartości surowców oraz pozycje wskaźników `head`/`tail` w buforze cyklicznym były identyczne jak w momencie zamknięcia.

Test 4: Odporność na awarie i wycieki pamięci (Signal Handling)

- **Cel:** Upewnienie się, że awaryjne zamknięcie programu nie pozostawia w systemie "wiszących" zasobów IPC (pamięci, semaforów) ani procesów zombie.
- **Przebieg:**
  1. Uruchomiono pełną symulację (1 Dyrektor + 4 Dostawców + 2 Pracowników).
  2. Wciśnięto kombinację `Ctrl+C` (SIGINT).
  3. W terminalu wpisano komendy `ipcs` (sprawdzenie zasobów IPC) oraz `ps -ef | grep dyrektor` (sprawdzenie procesów).
- **Wynik:**
  - Handler sygnału poprawnie wykonał procedurę `sprzatanie_i_wyjscie`.
  - Zastosowanie sygnału `SIGKILL` wobec procesów potomnych natychmiast odblokowało terminal (rozwiązanie problemu wiszącego `wait`).
  - Komenda `ipcs` potwierdziła, że segmenty pamięci i tablice semaforów zostały usunięte z systemu.

---

4. Linki do kodu (Implementacja wymagań)

Poniżej znajdują się odnośniki do kluczowych fragmentów kodu realizujących wymagane funkcje systemowe.

a. Tworzenie i obsługa plików

Operacje zapisu logów oraz serializacji stanu magazynu.

- **`fopen()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L131
- **`fprintf()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L133
- **`fclose()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L134
- **`write()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L76
- **`read()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L84

b. Tworzenie procesów

Zarządzanie cyklem życia procesów w modelu Unix.

- **`fork()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L111
- **`execl()`**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd

6e7f/src/dyrektor.cpp#L112

- `wait()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dostawca.cpp#L74
- `exit()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dostawca.cpp#L61

d. Obsługa sygnałów

Mechanizmy asynchronicznego powiadamiania procesów.

- `signal()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L72
- `kill()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L164

e. Synchronizacja procesów (Semafory System V)

Implementacja 10 semaforów do sterowania dostępem.

- `semget()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L79
- `semctl()` **(SETVAL)**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L23
- `semop()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/include/common.h#L106

g. Segmenty pamięci dzielonej

Współdzielenie stanu magazynu między procesami.

- `shmget()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L75
- `shmat()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L77
- `shmdt()`: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L21
- `shmctl()` **(IPC_RMID)**: https://github.com/olcixx999/fabryka_czekolady_so/blob/c4cae1f70ccfb9ed32a116333b2373469afd6e7f/src/dyrektor.cpp#L22