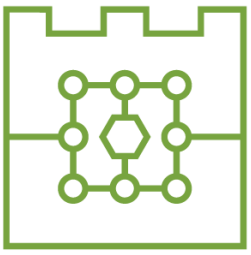

Politechnika Krakowska
Wydział Informatyki
i Telekomunikacji

Sprawozdanie – projekt Systemy Operacyjne
Temat 3: Fabryka

Tymon Gontarz – 152689 – Informatyka NST – rok 2. – semestr 3.

1. Treść zadania i wstępne założenia.

Treść zadania 3.

Fabryka posiada 2 stanowiska produkcyjne A i B. Na każdym stanowisku składane są wyroby z podzespołów X, Y i Z. Podzespoły przechowywane są w magazynie o pojemności M jednostek. Podzespół X zajmuje jedną jednostkę magazynową, podzespół Y dwie, a podzespół Z trzy jednostki. Podzespoły pobierane są z magazynu, przenoszone na stanowisko produkcyjne i montowane. Z podzespołów X, Y i Z po ich połączeniu powstaje jeden produkt, po czym pobierane są następne podzespoły z magazynu. Jednocześnie trwają dostawy podzespołów do fabryki. Podzespoły pochodzą z 3 niezależnych źródeł i dostarczane są w nieokreślonych momentach czasowych. Fabryka przyjmuje do magazynu maksymalnie dużo podzespołów dla zachowania płynności produkcji.

Magazyn kończy pracę po otrzymaniu polecenia_1 od dyrektora. Fabryka kończy pracę po otrzymaniu polecenia_2 od dyrektora. Fabryka i magazyn kończą pracę jednocześnie po otrzymaniu polecenia_3 od dyrektora – aktualny stan magazynu zapisany w pliku, po ponownym uruchomieniu stan magazynu jest odtwarzany z pliku. Fabryka i magazyn kończą pracę jednocześnie po otrzymaniu polecenia_4 od dyrektora – stan magazynu nie jest zapamiętany, po ponownym uruchomieniu magazyn jest pusty.

Napisz program dla procesów dyrektor oraz dostawca i monter reprezentujących odpowiednio: dostawców produktów X, Y i Z oraz pracowników na stanowiskach A i B.

Wstępne założenia

Proces dyrektor – proces główny,

Pozostałe procesy (dostawcy, monterzy) – procesy pochodne przy pomocy **fork()**

Komunikacja między procesami odbywa się przy pomocy:

- Pamięci dzielonej
- Kolejki komunikatów

2. Opis pracy nad projektem, przypadki użycia, testy

Opis pracy

Pierwszym etapem pracy nad projektem było opracowanie funkcji/plików pomocniczych (takich jak **defines.h** czy **ipc_utils.h/c**). Następnie przygotowany został „punkt wejściowy programu". Samo uruchamianie procesów pochodnych (i odpowiednich dla nich funkcji, na momenty wstępny pustych), nie stanowiło problemu. Dodatkowo przygotowane było definiowanie pamięci dzielonej i sprawdzenie, czy jest ona poprawnie wykorzystywana w procesach pochodnych.

Kolejnym etapem po opracowaniu podstaw (etap pierwszy), było opracowanie przesyłanie komunikatów między procesem głównym (dyrektor), a procesami pochodnymi. Do przesyłania komunikatów wykorzystano kolejki **mqd_t**. Samo zdefiniowanie kolejek i ich obsługa wewnątrz odpowiednich procesów nie stanowiła problemów, jednak problematyczne okazało się „nasłuchiwanie" przez procesy pochodne.

Problem został rozwiązany przy pomoc atrybutu kolejki **O_NONBLOCK**, dzięki któremu pętla główna nie jest blokowana w przypadku napotkania pustej kolejki. W przypadku pustej kolejki, pętla kontynuuje swoje działanie, a kolejka jest sprawdzana ponownie przy kolejnej iteracji pętli.

Następnym etapem było dokładne określenie działania wszystkich funkcji procesów.

W pierwszej kolejności proces **Dyrektora** – odpowiednia obsługa sygnałów, przesyłanie komunikatów do odpowiednich kolejek.

Następnie **Magazyn** – definiowanie danych magazynu (dane domyślne, lub czytanie z pliku tekstowego), obsługa otrzymanych komunikatów, wypisywanie stanu magazynu do dokumentu przy zamknięciu magazynu.

Potem przyszła kolej na **Dostawców i Monterów** – pobieranie/przesyłanie podzespołów do magazynu przy pomocy pamięci wspólnej, obsługa poleceń Dyrektora (w przypadku Monterów), weryfikacja czy magazyn jest otwarty (w przypadku Dostawców).

Wszystkie główne procesy są wykonywane w pętlach i synchronizowane przy pomocy odpowiedniego blokowania/odblokowywania semafora pamięci wspólnej.

Na tym etapie, sam program wykonywał wszystkie polecenia z treści zadania. Została jednak ew. optymalizacja, opisanie kodu komentarzami, poprawki (dodanie makr wrappujących pewne funkcje jak np. mqsend(), w celu weryfikacji poprawnego działania). Dane programu i jego stan były jednak nadal wyświetlane w terminalu, co utrudniało przesyłanie sygnałów do głównego procesu.

Ostatnim i najbardziej problematycznym etapem było przygotowanie interfejsu uruchamianego w zewnętrznym oknie.

Do obsługi interfejsu wykorzystana była **biblioteka ncurses** (pozwalająca na tworzenie UI wewnątrz terminalu oraz **xterm** (które pozwoliło na uruchomienie drugiego pobocznego terminala). Interfejs jest obsługiwany przez oddzielny plik wykonawczy uruchamiany z procesu pochodnego. Wewnątrz procesu pochodnego wywoływana jest komenda uruchamiająca zewnętrzny terminal **xterm**, w którym wywoływany jest plik wykonawczy interfejsu.

Dzięki temu sam interfejs wyświetla się w zewnętrznym oknie, natomiast możliwe jest wysyłanie sygnałów do głównego procesu z poziomu głównego terminalu.

Kluczowe przypadki użycia

1. **Dostawa podzespołów do magazynu**

aktorzy: Magazyn, Dostawcy (X, Y, Z)

opis: Dostawcy dostarczają podzespół do Magazynu w losowych momentach, pod warunkiem, że w Magazynie jest wystarczająca ilość miejsca (i że magazyn jest **otwarty**).

warunek końcowy: po przyjęciu podzespołu aktualizowana jest ich ilość w Magazynie i pojemność magazynu

sposób wykonania: aktualizowanie danych w pamięci wspólnej procesów

2. **Pobranie podzespołów przez monterów**

aktorzy: Magazyn, Monterzy (A, B)

opis: Monterzy pobierają podzespoły z Magazynu w losowych momentach, pod warunkiem, że w Magazynie jest wystarczająca ilość podzespołów (i że magazyn jest **otwarty**).

warunek końcowy: po pobraniu podzespołów aktualizowana jest ich ilość w Magazynie i pojemność magazynu

sposób wykonania: aktualizowanie danych w pamięci wspólnej procesów

3. **Zapełnienie magazynu / magazyn pusty**

aktorzy: Magazyn, Monterzy (A, B), Dostawcy (X, Y, Z)

opis: W momencie gdy magazyn zostaje zapełniony (lub nie ma wystarczająco miejsca na podzespół), odrzucona zostaje dostawa od Dostawcy. W momencie gdy w magazynie brakuje podzespołów, odrzucone zostaje pobranie ich przez Monterów.

warunek końcowy: odrzucenie dostarczenia lub pobrania podzespołów z magazynu

sposób wykonania: warunki sprawdzające stan danych w pamięci wspólnej

4. **Wstrzymanie pracy magazynu (polecenie_1)**

aktorzy: Magazyn, Dostawcy (X, Y, Z), Dyrektor

opis: W momencie gdy Dyrektor wyda polecenie_1, Magazyn zostaje zamknięty. Dostawcy kończą pracę ze względu na brak możliwości wykonania dostaw.

warunek końcowy: blokowana możliwość dostaw i pobrań, Dostawcy kończą pracę

sposób wykonania: Dyrektor odbiera sygnał polecenie_1, następnie przesyła polecenie kolejką komunikatów do Magazynu. W momencie gdy Magazyn odbierze komunikat, aktualizuje dane w pamięci dzielonej (magazyn zamknięty, bool = FALSE). Po sprawdzeniu stanu magazynu w pamięci dzielonej przez Dostawców, Dostawcy kończą pracę (zakończenie pętli działania w procesach).

5. **Wstrzymanie pracy fabryki (polecenie_2)**

aktorzy: Monterzy (A, B), Dyrektor

opis: W momencie gdy Dyrektor wyda polecenie_2, Monterzy kończą pracę.

warunek końcowy: zakończenie pracy Monterów, brak pobierania podzespołów z magazynu

sposób wykonania: Dyrektor odbiera sygnał polecenie_2, następnie przesyła polecenie kolejką komunikatów do Monterów. W momencie gdy Monterzy odbierą komunikat, zakończona zostaje główna pętla działania Monterów, procesy Monter kończą pracę

6. **Wstrzymanie pracy magazynu i fabryki, zapisanie stanu magazynu (polecenie_3)**

aktorzy: Magazyn, Monterzy (A, B), Dostawcy (X, Y, Z), Dyrektor

opis: W momencie gdy Dyrektor wyda polecenie_3, Magazyn zostaje zamknięty, Monterzy kończą pracę, Dostawcy kończą pracę. Stan magazynu zostaje zapisany.

warunek końcowy: blokowana możliwość dostaw i pobrań, Dostawcy kończą pracę, Monterzy kończą pracę, zapisano stanu magazynu

sposób wykonania: Dyrektor odbiera sygnał polecenie_3, następnie przesyła polecenie kolejką komunikatów do Magazynu i Monterów. W momencie gdy Magazyn odbierze komunikat, aktualizuje dane w pamięci dzielonej (magazyn zamknięty, bool = FALSE) oraz zapisuje stan magazynu do pliku tekstowego. W momencie gdy Monterzy odbierą komunikat, zakończona zostaje główna pętla działania Monterów, procesy Monter kończą pracę. Po sprawdzeniu stanu magazynu w pamięci dzielonej przez Dostawców, Dostawcy kończą pracę (zakończenie pętli działania w procesach).

7. **Wstrzymanie pracy magazynu i fabryki, bez zapisania stanu magazynu (polecenie_4)**

aktorzy: Magazyn, Monterzy (A, B), Dostawcy (X, Y, Z), Dyrektor

opis: W momencie gdy Dyrektor wyda polecenie_4, Magazyn zostaje zamknięty, Monterzy kończą pracę, Dostawcy kończą pracę. Stan magazynu nie zostaje zapisany.

warunek końcowy: blokowana możliwość dostaw i pobrań, Dostawcy kończą pracę, Monterzy kończą pracę

sposób wykonania: Dyrektor odbiera sygnał polecenie_4, następnie przesyła polecenie kolejką komunikatów do Magazynu i Monterów. W momencie gdy Magazyn odbierze komunikat, aktualizuje dane w pamięci dzielonej (magazyn zamknięty, bool = FALSE). W momencie gdy Monterzy odbierą komunikat, zakończona zostaje główna pętla działania Monterów, procesy Monter kończą pracę. Po sprawdzeniu stanu magazynu w pamięci dzielonej przez Dostawców, Dostawcy kończą pracę (zakończenie pętli działania w procesach).

Testy

Przed skompilowaniem głównego programu wykonawczego, wykonane zostają 3 testy. W przypadku, gdy wszystkie 3 testy się powiodą – główny program wykonawczy zostaje skompilowany. W przypadku niepowodzenia, główny program wykonawczy nie jest skompilowany.

1. **TEST 1 – poprawne operowanie na dynamicznej liście PID**

Program testowy deklaruje i inicjuje dynamiczną listę PID, następnie weryfikuje dane początkowe w liście.

Następnie do listy jest dodawany nowy PID, ponownie weryfikowana jest poprawność danych zawartych w liście.

Na koniec lista zostaje uwolniona/zresetowana, ponownie weryfikowana jest poprawność danych zawartych w liście.

Jeżeli wszystkie etapy przeszły poprawnie, TEST 1 zakończył się sukcesem.

2. **TEST 2 – poprawne działanie funkcji odpowiedzialnych za interfejs**

Program testowy symuluje wypisanie listy PID do pliku tekstowego, następnie deklaruje i inicjuje strukturę listy PID'ów i czyta plik tekstowy, aby wypełnić listę PID'ów przykładowymi danymi. W trakcie tych operacji, weryfikowane jest czy plik tekstowy został poprawnie otwarty, czy dane zostały poprawnie zapisane i odczytane.

Następnie inicjowane jest ncurses, tworzone jest testowe okno ncurses. Weryfikowane jest, czy okno zostało poprawnie utworzone.

Kolejnym krokiem jest wprowadzenie danych do okna ncurses czytając PID'y z listy PID'ów. Przy każdym wprowadzeniu danych do okna weryfikowane jest, czy operacja się powiodła.

Następnie okno jest zawierane w ramce. Weryfikowane jest, czy operacja box() się powiodła.

Na koniec usuwane jest okno i kończona jest praca ncurses.

Jeżeli wszystkie etapy przeszły poprawnie, TEST 2 zakończył się sukcesem.

3. **TEST 3 – poprawna obsługa sygnałów**

Program przypisuje funkcję odpowiedzialną za obsługę sygnałów do przykładowego sygnału (jeden z sygnałów obsługiwanych przez główny program). Następnie do programu wysyłany jest sygnał, który ma być obsłużony.

Po przyjęciu sygnału, weryfikowana jest zawartość danych, które mają być zmodyfikowane na podstawie otrzymanego sygnału.

Jeżeli wszystkie etapy przeszły poprawnie, TEST 3 zakończył się sukcesem.

4. **TEST 4 (dodatkowy) – stress-test, uruchomienie głównego programu po skompilowaniu przy wyłączonych funkcjach sleep()**

Główny program wykonawczy jest skompilowany ze zmienną preprocesora USE_SLEEP = 0, wyłączając wszystkie funkcje sleep() w programie. Sprawdzana jest stabilność programu i synchronizacji.

Test wykonywany jest dodatkowo, poza głównym poleceniem testowania i kompilacji (oddzielny plik wykonawczy nosleep_fabryka).

3. Opis i podsumowanie kodu

Główny katalog projektu

Katalog główny zawiera pliki odpowiedzialne za zadania wykonywane przez skrypty bashowe oraz podkatalogi zgodnie z zastosowaniem plików.

- Bezpośrednio w katalogu głównym

Pliki shell odpowiedzialne za wykonywanie poszczególnych zadań:

- **build.sh** – kompilowanie całego programu
- **build_w_tests.sh** – kompilowanie całego programu pod warunkiem powodzenia się wszystkich testów
- **verify_file_state.sh –** dodatkowy plik sprawdzający, które pliki zostały oflagowane jako ukończone/nieukończone (FINISHED/UNFINISHED)

Dodatkowo zawiera README.md oraz changelog.

- Katalog **./src**

Zawiera wszystkie pliki źródłowe programu.

- Katalog **./bin**

Zawiera pliki wykonawcze projektu oraz pliki wykorzystywane przez program w trakcie jego pracy.

- Katalog **./doc**

Dokumentacja związana z projektem, dodatkowa dokumentacja pomocnicza.

- Katalog **./.vscode**

Pliki konfiguracyjne środowiska VSCode.

Kod źródłowy

Cały kod źródłowy znajduje się w katalogu ./**src**, jego zawartość jest natomiast podzielona na 4 „segmenty”.

- Pliki bezpośrednio zawarte w **./src**

Pliki źródłowe zawierające funkcje i struktury pomocnicze wykorzystywane przez resztę programu. Dodatkowo plik **defines.h** – załącza wszystkie niezbędne biblioteki systemowe, weryfikuje czy program jest kompilowany/uruchamiany wyłączenie na Linuxie, definiuje podstawowe domyślne/konfiguracyjne zmienne.

- Pliki zawarte w podkatalogu **./core**

Pliki źródłowe odpowiadające za właściwe działanie programu. Punkt rozpoczynający działanie całego programu zawarty jest w **enty.c** -> następnie z tego punku uruchamiane są wszystkie pozostałe procesy pochodne. Wewnątrz **entry.c** deklarowana/definiowana jest pamięć wspólna, poszczególne kolejki komunikatów otwierane są wewnątrz właściwych pętli działania każdego procesu. Dodatkowo, wewnątrz „punktu wejściowego” uruchamiany jest proces odpowiedzialny za otwarcie interfejsu graficznego programu w zewnętrznym oknie (korzystając z **xterm**).

Dyrektor (manager.h / manager.c) – odpowiedzialne za główną pętlę działania procesu głównego. Funkcja dyrektora uruchamiana jest bezpośrednio w głównym procesie. Z tego poziomu odbierane są sygnały wysyłane do programu i wysyłane są komunikaty do pozostałych procesów pochodnych. Wewnątrz procesu dyrektora, tworzone i otwierane są wszystkie kolejki komunikatów, dyrektor czeka na

odczytanie sygnału i w zależności od danego sygnału, wysyła odpowiednie polecenia do procesów pochodnych.

Magazyn (warehouse.h / warehouse.c) – odpowiedzialne za pętlę działania procesu magazynu. Odpowiedzialny za inicjowanie wartości struktury magazynu, czytanie/pisanie rejestru magazynu do pliku tekstowego. Na podstawie stanu struktury magazynu (określanej z tego poziomu) – określa czy działanie powinny zakończyć procesy dostawców (w momencie „zamknięcia" magazynu, dostawcy przestają dostarczać podzespoły). Wewnątrz procesu magazynu otwierana jest kolejka komunikatów magazynu, główna pętla „nasłuchuje", czy dyrektor przesłał polecenie, następnie wykonuje działania zgodnie z dostarczonym poleceniem.

Dostawcy (supplier.h / supplier.c) – odpowiedzialne za pętle działania każdego procesu dostawcy (oddzielny proces dla każdego podzespołu X, Y, Z). Odpowiedzialny za dostarczanie detali do magazynu (przy okazji aktualizując pojemność magazynu przy każdej dostawie). W przypadku zapełniania magazynu, dostawa nie jest wykonana. W momencie zamknięcia magazynu, dostawcy kończą działanie.

Monterzy (manufacturer.h / manufacturer.c) – odpowiedzialne za pętle działania każdego procesu montera (oddzielny proces dla A i B). Monterzy pobierają części z magazynu, przy okazji aktualizując stan pojemności magazynu. W momencie zamknięcia magazynu (lub w przypadku braku detali), monterzy nie pobierają detali z magazynu i czekają na aktualizację danych w magazynie. Wewnątrz procesu otwierana jest odpowiednia kolejka komunikatów monterów, główna pętla „nasłuchuje", czy dyrektor przesłał polecenie, następnie wykonuje działania zgodnie z dostarczonym poleceniem.

- Pliki zawarte w podkatalogu **./ui**

Pliki źródłowe (ui_terminal.h / ui.terminal.c) odpowiedzialne za obsługę interfejsu graficznego, zawierającego aktualizujące się dane dotyczące stanu magazynu oraz ID wszystkich procesów (proces główny, do którego przesyłane są sygnały, oraz wszystkich procesów pochodnych).

Interfejs wykorzystuję bibliotekę ncurses, pozwalającą na utworzenie UI wewnątrz terminalu.

Proces interfejsu jest uruchamiany z oddzielnego pliku wykonywalnego (**ui_fabryka**), przesyłając id dzielonej pamięci i semafora przy pomocy argumentów funkcji main. Następnie wewnątrz procesu UI dostęp do pamięci wspólnej jest otrzymany przy pomocy argumentów przyjętych przez funkcję main.

Wewnątrz głównej pętli procesu interfejsu odświeżane są dane interfejsu. Dodatkowo, otwierana jest kolejka komunikatów interfejsu. Główna pętla „nasłuchuje", czy dyrektor przesłał polecenie, następnie wykonuje działania zgodnie z dostarczonym poleceniem

- Pliki zawarte w podkatalogu **./tests**

Zawiera **plik źródłowy (run_tests.c)** odpowiedzialny za testy wykonywane przez skompilowaniem właściwego programu. Po uruchomieniu głównej komendy kompilującej, najpierw kompilowany i wykonywany jest program odpowiedzialny za testy. Jedynie w przypadku poprawnego wykonania wszystkich testów, skompilowane zostaną: program właściwy, program odpowiedzialny za interfejs.