Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Systemy operacyjne

Zarządzanie pamięcią
Raport

Zdający:
Jakub Sikora

Prowadzący:
mgr. inż. Aleksander Pruszkowski

Warszawa, 8 stycznia 2019

Spis treści

1. Treść zadania

Zgodnie z wytycznymi podanymi na stronie "Ćwiczenie 3 Synchronizacja procesów z wykorzystaniem semaforów" , napisz usługę „chat” dającą użytkownikom możliwość komunikacji między sobą (w obrębie jednej maszyny). Załóż, że niektórzy wśród użytkowników mają podwyższone prawa i ich wiadomości „wskakują” na początek kolejki rozsyłania wiadomości do pozostałych użytkowników. Załóż, że wszyscy użytkownicy komunikują się poprzez system w wyłącznie jednym temacie. Przemyśl metodę testowania powstałego systemu, w szczególności zwróć uwagę na pokazanie w działaniu równoczesnego „chatowania” obu grup użytkowników – w tym udowodnienia, że system faktycznie obsługuje wiadomości uwzględniając priorytety. Załóż, że wytworzone przez Ciebie programy testowe będą działały automatycznie generując testowe fikcyjne wiadomości a jeden z nich będzie te wiadomości pokazywał na konsoli.

2. Realizacja rozwiązania

Usługę "chatowania" zrealizowałem za pomocą zestawu semaforów systemowych, chroniących jednoczesnego dostępu do dwóch buforów cyklicznych zaalokowanych w pamięci współdzielonej.

2.1. Bufory cykliczne

Wiadomości przekazywane w ramach usługi "chatu" przechowywane były w buforach cyklicznych. Każdy z buforów alokował pamięć na pewną ilość elementów dalej oznaczaną `MAX` oraz na dwa wskaźniki: odczytu i zapisu. Proces zapisujący nową wiadomość zapisuje ją pod adresem wskazywanym przez wskaźnik zapisu a następnie przesuwa go o jedną pozycję do przodu. Jeśli w momencie dodawania nowej wiadomości, wskaźnik zapisu znajdzie się na ostatniej pozycji bufora(indeks `MAX-1`), proces przesuwa go na pozycję pierwszą o indeksie zero. Analogicznie, proces odczytujący wiadomość z bufora pobiera ją z miejsca wskazywanego przez wskaźnik odczytu a następnie przesuwa go o jedną pozycję. Mechanizmy synchronizacji powinny zapewnić że tylko jeden proces może aktualnie pracować na buforze cyklicznym oraz muszą zabezpieczyć przed wyprzedzeniem wskaźnika zapisu przez wskaźnik odczytu.

W pojedynczym buforze cyklicznym, nie jest możliwe poprawne obsłużenie mechanizmu priorytetowego pobierania wiadomości specjalnych, dlatego też w przedstawionym rozwiązaniu zaimplementowałem dwa bufory z czego pierwszy obsługuje tylko wiadomości wysłane przez procesy uprzywilejowane a drugi tylko wiadomości zwykłe.

Bufory cykliczne zostały zaimplementowane w pamięci współdzielonej. Inicjalizację pamięci wykonałem za pomocą funkcji `shmget` a wskaźnik na zaalokowaną pamięc uzyskałem za pomocą procedury `shmat`.

2.2. Semafory

Aby umożliwić synchronizację procesów wysyłających (producentów) oraz odbierającego (konsumenta), skorzystałem z zestawu semaforów oferowanych przez system operacyjny `Linux`. Do poprawnej obsługi problemu, należało skorzystać z pięciu semaforów. Para semaforów (w przedstawionym rozwiązaniu oznaczonych numerami 0 i 1) chroni przed niepoprawnym dostępem do bufora cyklicznego. Semafor o numerze 0 inicjowany wartością `MAX` wskazuje ile jest wolnych slotów w buforze. Wartość ta jest dekrementowana przez producentów oraz dekrementowana przez konsumenta. Cechą semaforów jest fakt, że próba dekrementacji semafora, którego wartośc wynosi 0, spowoduje zawieszenie procesu do momentu zwolnienia się miejsca w buforze (czyli do momentu gdy konsument zinkrementuje wartość tej zmiennej poprzez pobranie elementu z bufora, zwalniając przy tym slot na następną wiadomość). Semafor o numerze 1 działa analogicznie do poprzedniego. Inicjowany wartością 0, zlicza ile elementów jest gotowych do pobrania przez konsumenta. Semafor ten zabezpiecza przed odczytem pustego bufora. Konsument próbując odczytać nową wiadomość z pustego bufora, zostanie zawieszony do momentu gdy któryś z producentów włoży do bufora nowy element. Analogicznie działają semafory o numerach 2 i 3. Pilnują one przed niepoprawnym dostępem do bufora trzymającego dane

priorytetowe.

Ostatni semafor oznaczony numerem 4, zabezpiecza oba bufory przed jednoczesnym dostępem wielu procesów. Inicjowany wartością 1, pilnuje aby tylko jeden proces mógł edytować dowolny z buforów.

Zbiór semaforów uzyskałem za pomocą polecenia `semget` a operacje inkrementacji i dekrementacji wykonywałem za pomocą procedury `semop`.

2.3. Przykład producenta

Przykładowa operacja wstawiania nowej wiadomości do bufora wygląda w następujący sposób:

```
decrease(semid, 0);
decrease(semid, 4);
buf[indexZ] = arg;
indexZ = (indexZ+1) % MAX;
increase(semid, 1);
increase(semid, 4);
```

W pierwszej kolejności, proces wstawiający sprawdza czy jest miejsce w buforze poprzez dekrementację wartości semafora 0. Producent przechodząc do następnego polecenia, niejako rezerwuje sobie wolny slot w buforze. W linijce drugiej, proces wstrzymuje się do momentu aż będzie mógł wejść do sekcji krytycznej i będzie miał dostęp do buforów na wyłączność. Linijki 3 i 4 wykonują operację zapisu do bufora cyklicznego i inkrementację wskaźnika zapisu. W ostatnich dwóch linijkach proces zwiększa ilość elementów gotowych w buforze oraz zwiększa semafor 4, informując inne procesy że wyszedł z sekcji krytycznej.

2.4. Przykład konsumenta

Przykładowa operacja konsumenta jest nieco bardziej złożona, ponieważ musi brać pod uwagę istnienie dwóch buforów:

```
while(1){
    decrease(semid, 4);
    if(semctl(semid, 3, GETVAL) != 0){
        decrease(semid, 3);
        printf("%5d\n", buf[indexOpri])
        indexOpri = (indexOpri+1)%MAX;
        increase(semid, 2);
        increase(semid, 4);
        continue;
    }
    else{
        if(semctl(semid, 1, GETVAL) != 0){
            decrease(semid, 1);
            printf("%5d\n", buf[indexO])
            indexO = (indexO+1)%MAX;
            increase(semid, 0);
            increase(semid, 4);
        }
        else {
            increase(semid, 4);
        }
    }
}
```

Program pojedynczego konsumenta z zadaną częstotliwością skanuje sekcję krytyczną w poszukiwaniu nowych wiadomości. Konsument wchodzi do sekcji krytycznej poprzez dekrementację semafora 4. Wiedząc że tylko jeden proces może się znajdować w sekcji krytycznej oraz że to konsument się w niej znajduje, może on sprawdzić wartości semaforów bez obaw że ich wartość może się zmienić bez jego wiedzy. W pierwszej kolejności proces sprawdza czy są jakieś wiadomości priorytetowe. Jeśli nie ma żadnych, proces sprawdza czy są jakieś wiadomości zwyczajne. Jeżeli w danym buforze są wiadomości i konsument zdecydował się wyjąć z niego wiadomość, proces dekrementuje semafor który przechowuje informację o ilości zajętych slotów. Zużycie wiadomości następuje poprzez wypisanie treści oraz przesunięcie wskaźnika odczytu. Ostatecznie inkrementowane są wartości semaforów oznaczających ilość pustych slotów w buforze oraz semafor 4 implementujący wzajemne wykluczanie sygnalizując wyjście z sekcji krytycznej.

3. Testowanie rozwiązania

3.1. Programy testowe

Aby przeprowadzić testy przedstawionego rozwiązania przygotowałem kilka programów w języku C. Każdy z programów sprawdza przy uruchomieniu czy pod danym stałym kluczem zainicjalizowane zostały już mechanizmy pamięci współdzielonej. Jeśli nie, same dokonują inicjalizacji.

3.1.1. Program czytający - `rx`

Program czytający `rx` pełni w przedstawionym problemie rolę konsumenta. Imituje on konsolę na której wypisywane są wiadomości przesłane przez producentów. Program co okres $1s$, sprawdza czy są jakieś wiadomości do odczytania zgodnie z mechanizmem przedstawionym w sekcji 2.4.

3.1.2. Programy wysyłające - `tx`, `spamtx`, `bursttx`

Programy wysyłające w przedstawionym problemie pełni rolę producenta. Wszystkie przedstawione programy działają zgodnie z zasadą przedstawioną w sekcji 2.3. Różnią się tylko częstotliwością działania. Program `tx` wysyła łącznie dziesięć wiadomości w odstępach 2 sekund. Program `spamtx` również wysyła dziesięć wiadomości, jednak wysyła je bez odstępów czasowych. Ostatni program `bursttx` wysyła wiadomości w sposób podobny do `spamtx`, jednak ilość wysłanych wiadomości wynosi 500. Jest to ilość wystarczająca do przepełnienia bufora oraz prezentacji działania rozwiązania w takich okolicznościach. Programy te jako argument wywołania pobuierją numer, który będzie wysyłany jako wiadomość.

3.1.3. Programy priorytetowe - `px`, `spampx`, `burstpx`

Programy priorytetowe działają na tej samej zasadzie na jakiej działają zwykłe programy wysyłające opisane w podsekcji 3.1.2. Różnią się tylko destynacją wiadomości. Wysyłają one wyprodukowane wiadomości do bufora priorytetowego `buf_pri`.

3.2. Scenariusze testowe

W ramach zadania przygotowałem kilka scenariuszy testowych prezentujących poprawność zaimplementowanego rozwiązania.

3.2.1. Odczyt z pustego bufora

W pierwszym teście dokonałem próby odczytu z pustego bufora. W tym celu uruchomiłem tylko program czytający `rx`.

```
sikora@dell:~/Documents/Dev/Soi/SOI-laboratorium/soi3$ ./rx
Brak wiadomości w buforze.
Brak wiadomości w buforze.
Brak wiadomości w buforze.
Brak wiadomości w buforze.
```

W każdej iteracji, program wchodził do sekcji krytycznej i sprawdzał czy w obu buforach są jakieś dane do wyświetlenia. Za każdym razem program żadnych danych nie znajdywał, co sygnalizował stosownym komunikatem. Brak wiadomości w buforach nie powodował zawieszenia programu czytającego.

3.2.2. Zachowanie kolejności wiadomości

W kolejnym teście uruchomiłem trzy procesy nadawcze `tx` oraz jeden proces odbiorczy. W ramach tego testu sprawdziłem czy elementy o równym priorytecie wychodzą z bufora w takiej samej kolejności w jakiej do niego weszły.

Test wykonałem za pomocą skryptu `test1.sh`.

```
./tx 200&
sleep 0.1
./tx 300&
sleep 0.1
./tx 400&
```

Wynik testu znajduje się poniżej.

```
./rx                                ./test1.sh
2018-12- 5  0:28:43   200           2018-12- 5  0:28:43: 0. Wysłano: 200
2018-12- 5  0:28:44   300           2018-12- 5  0:28:43: 0. Wysłano: 300
2018-12- 5  0:28:45   400           2018-12- 5  0:28:43: 0. Wysłano: 400
2018-12- 5  0:28:46   200           2018-12- 5  0:28:45: 1. Wysłano: 200
2018-12- 5  0:28:47   300           2018-12- 5  0:28:45: 1. Wysłano: 300
2018-12- 5  0:28:48   400           2018-12- 5  0:28:45: 1. Wysłano: 400
2018-12- 5  0:28:49   200           2018-12- 5  0:28:47: 2. Wysłano: 200
2018-12- 5  0:28:50   300           2018-12- 5  0:28:47: 2. Wysłano: 300
2018-12- 5  0:28:51   400           2018-12- 5  0:28:47: 2. Wysłano: 400
2018-12- 5  0:28:52   200           2018-12- 5  0:28:49: 3. Wysłano: 200
2018-12- 5  0:28:53   300           2018-12- 5  0:28:49: 3. Wysłano: 300
2018-12- 5  0:28:54   400           2018-12- 5  0:28:49: 3. Wysłano: 400
2018-12- 5  0:28:55   200           2018-12- 5  0:28:51: 4. Wysłano: 200
2018-12- 5  0:28:56   300           2018-12- 5  0:28:51: 4. Wysłano: 300
2018-12- 5  0:28:57   400           2018-12- 5  0:28:51: 4. Wysłano: 400
2018-12- 5  0:28:58   200           2018-12- 5  0:28:53: 5. Wysłano: 200
2018-12- 5  0:28:59   300           2018-12- 5  0:28:53: 5. Wysłano: 300
2018-12- 5  0:29: 0   400           2018-12- 5  0:28:53: 5. Wysłano: 400
2018-12- 5  0:29: 1   200           2018-12- 5  0:28:55: 6. Wysłano: 200
2018-12- 5  0:29: 2   300           2018-12- 5  0:28:55: 6. Wysłano: 300
2018-12- 5  0:29: 3   400           2018-12- 5  0:28:55: 6. Wysłano: 400
2018-12- 5  0:29: 4   200           2018-12- 5  0:28:57: 7. Wysłano: 200
2018-12- 5  0:29: 5   300           2018-12- 5  0:28:57: 7. Wysłano: 300
2018-12- 5  0:29: 6   400           2018-12- 5  0:28:57: 7. Wysłano: 400
2018-12- 5  0:29: 7   200           2018-12- 5  0:28:59: 8. Wysłano: 200
2018-12- 5  0:29: 8   300           2018-12- 5  0:28:59: 8. Wysłano: 300
2018-12- 5  0:29: 9   400           2018-12- 5  0:28:59: 8. Wysłano: 400
2018-12- 5  0:29:10   200           2018-12- 5  0:29: 1: 9. Wysłano: 200
2018-12- 5  0:29:11   300           2018-12- 5  0:29: 1: 9. Wysłano: 300
2018-12- 5  0:29:12   400           2018-12- 5  0:29: 1: 9. Wysłano: 400
```

Listing powyżej jednoznacznie udowadnia że zaimplementowane rozwiązanie poprawnie zachowuje kolejność wiadomości przychodzących.

3.2.3. Priorytet wiadomości

Kolejny test miał na celu sprawdzić czy przedstawione rozwiązanie poprawnie interpretuje wiadomości z priorytetem i obsługuje je jako pierwsze. W tym celu napisałem kolejny skrypt o nazwie `test2.sh`.

```
./tx 200&
./tx 300&
./tx 400&
sleep 1.5s
./px 9999&
```

W tym teście, procesy zwykłe wysyłały wiadomość co dwie sekundy a proces priorytetowy wysyłał wiadomości co trzy sekundy. Proces konsumencki skanował wiadomości co jedną sekundę. Wynik testu znajduje się poniżej.

```
./rx
2018-12- 5  0:42:25   200
2018-12- 5  0:42:26   400
2018-12- 5  0:42:27  9999
2018-12- 5  0:42:28   300
2018-12- 5  0:42:29   200
2018-12- 5  0:42:30  9999
2018-12- 5  0:42:31   400
2018-12- 5  0:42:32   300
2018-12- 5  0:42:33  9999
2018-12- 5  0:42:34   200
2018-12- 5  0:42:35   400
2018-12- 5  0:42:36  9999
2018-12- 5  0:42:37   300
2018-12- 5  0:42:38   200
2018-12- 5  0:42:39  9999
2018-12- 5  0:42:40   400
2018-12- 5  0:42:41   300
2018-12- 5  0:42:42  9999
2018-12- 5  0:42:43   200
2018-12- 5  0:42:44   400
2018-12- 5  0:42:45  9999
2018-12- 5  0:42:46   300
Brak wiadomości w buforze.
2018-12- 5  0:42:48  9999
Brak wiadomości w buforze.
Brak wiadomości w buforze.
2018-12- 5  0:42:51  9999
Brak wiadomości w buforze.
Brak wiadomości w buforze.
2018-12- 5  0:42:54  9999
```

```
./test2.sh
2018-12- 5  0:42:25: 0. Wysłano: 200
2018-12- 5  0:42:25: 0. Wysłano: 400
2018-12- 5  0:42:25: 0. Wysłano: 300
2018-12- 5  0:42:27: 0. Wysłano: 9999
2018-12- 5  0:42:27: 1. Wysłano: 200
2018-12- 5  0:42:27: 1. Wysłano: 400
2018-12- 5  0:42:27: 1. Wysłano: 300
2018-12- 5  0:42:29: 2. Wysłano: 200
2018-12- 5  0:42:29: 2. Wysłano: 400
2018-12- 5  0:42:29: 2. Wysłano: 300
2018-12- 5  0:42:30: 1. Wysłano: 9999
2018-12- 5  0:42:31: 3. Wysłano: 200
2018-12- 5  0:42:31: 3. Wysłano: 400
2018-12- 5  0:42:31: 3. Wysłano: 300
2018-12- 5  0:42:33: 2. Wysłano: 9999
2018-12- 5  0:42:33: 4. Wysłano: 200
2018-12- 5  0:42:33: 4. Wysłano: 400
2018-12- 5  0:42:33: 4. Wysłano: 300
2018-12- 5  0:42:36: 3. Wysłano: 9999
2018-12- 5  0:42:39: 4. Wysłano: 9999
2018-12- 5  0:42:42: 5. Wysłano: 9999
2018-12- 5  0:42:45: 6. Wysłano: 9999
2018-12- 5  0:42:48: 7. Wysłano: 9999
2018-12- 5  0:42:51: 8. Wysłano: 9999
2018-12- 5  0:42:54: 9. Wysłano: 9999
```

Na podstawie znajomości częstotliwości z jakimi iterują dane procesy oraz znaczników czasowych, można jednoznacznie stwierdzić że zaimplementowane przeze mnie rozwiąznie pomyślnie obsługuje wiadomości priorytetowe. Nawet przy sporej ilości oczekujących wiadomości zwykłych, w pierwszej kolejności i tak są brane pod uwagę wiadomości priorytetowe.

3.2.4. Przepełnienie bufora

W tym teście sprawdziłem jak rozwiązanie radzi sobie z przepełnieniem bufora na wiadomości. W związku z tym uruchomiłem tylko proces wysyłający równocześnie 500 wiadomości zwykłych. Jako że maksymalna pojemność bufora została przeze mnie ustalona na 400, proces powinien zawiesić się przed wysłaniem 401 wiadomości. Przy poprawnej implementacji rozwiązania, proces powinien oczekiwać na konsumenta wiadomości i dopiero wtedy zacząć wysyłać nowe dane, gdy z proces czytający zacznie opróżniać bufor.

Test zrealizowałem za pomocą programu `bursttx` opisanego w sekcji 3.1.2.

```
./bursttx 200
2018-12- 5  1: 1:40: 0. Wysłano: 200
2018-12- 5  1: 1:40: 1. Wysłano: 200
2018-12- 5  1: 1:40: 2. Wysłano: 200
2018-12- 5  1: 1:40: 3. Wysłano: 200

⋮

2018-12- 5  1: 1:40: 395. Wysłano: 200
2018-12- 5  1: 1:40: 396. Wysłano: 200
2018-12- 5  1: 1:40: 397. Wysłano: 200
2018-12- 5  1: 1:40: 398. Wysłano: 200
2018-12- 5  1: 1:40: 399. Wysłano: 200
```

Zgodnie z oczekiwaniami, proces zatrzymał się po wysłaniu 400 wiadomości. Zaimplementowane skutecznie zapobiegło przepełnieniu bufora.

3.2.5. Zagłodzenie procesów

Ostatni test miał na celu sprawdzić czy nie występuje zjawisko głodzenia procesów. W tym scenariuszu sprawdzałem czy w przypadku w całości zapełnionego bufora, stopniowo opróżnianego przez konsumenta dostęp do nowopowstałego slotu będzie rodzielany pomiędzy oczekujące procesy. W tym celu uruchomiłem program czytający `rx` oraz dwóch nadawców `tx` z argumentami 200 i 300.

Poniżej znajduje się końcówka uzyskanych z eksperymentu logów nadawców.

```
./test6.sh

⋮

2018-12- 5  1: 4:20: 198. Wysłano: 300
2018-12- 5  1: 4:21: 229. Wysłano: 200
2018-12- 5  1: 4:22: 199. Wysłano: 300
2018-12- 5  1: 4:23: 230. Wysłano: 200
2018-12- 5  1: 4:24: 200. Wysłano: 300
2018-12- 5  1: 4:25: 231. Wysłano: 200
```

Na podstawie testu można stwierdzić że każdy z procesów jest równomiernie dopuszczany do sekcji krytycznej. Róznice w indeksach wiadomości wynikają z nierównego startu obu procesów.

3.3. Podsumowanie

Przedstawiony powyżej zestaw testów jednoznacznie potwierdza poprawność przedstawionego rozwiązania oraz że skutecznie rozwiązuje najpoważniejsze problemy synchronizacyjne w przypadku problemu konsumenta i wielu producentów o różnych priorytetach.