JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 11 (2019) 149-154
Wysłane: 2019-04-01
Przyjęte: 2019-04-05

Analiza porównawcza wybranych zagadnień programowania wymagających komunikacji międzyprocesowej i międzywątkowej

Kamil Wróbel*

Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Poniższy artykuł przedstawia porównanie mechanizmów komunikacji i synchronizacji oferowanych w aplikacjach wielowątkowych oraz analogicznych rozwiązań opartych o komunikację międzyprocesową. Porównanie teoretyczne zestawiono z praktycznymi klasycznymi problemami synchronizacji. Porównano dostępność mechanizmów komunikacji i synchronizacji w bibliotece Boost i w komunikacji międzyprocesowej IPC. Przedstawiono subiektywną ocenę stopnia trudności implementacji i porównanie wydajności.

Słowa kluczowe: wątki; procesy; BOOT; komunikacją międzyprocesowa; systemy operacyjne

*Autor do korespondencji.

Adres e-mail: wrobelkamilx@gmail.com*

Comparative analysis of selected programming issues requiring inter-process and inter-thread communication

Kamil Wrobel*

Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The following article presents a comparison of multithread and multiprocess communication and synchronization. The theoretical comparison was supported by the solutions of practical, classical synchronization problems. The availability of communication and synchronization mechanisms in the Boost library and in IPC interprocess communication was compared.A subjective assessment of the difficulty in implementation and applications performance was also presented.

Keywords: threads; processes; boost; IPC; operation systems

*Corresponding author.

E-mail address: wrobelkamilx@gmail.com*

1. Wstęp

Systemy operacyjne zostały stworzone po to aby stanowić połączenie pomiędzy warstwą sprzętową komputera a warstwą interfejsu użytkownika, która umożliwia mu dokonywanie interakcji z komputerem. W momencie uruchomienia programu jest tworzony proces czyli aktywność, która jest odpowiedzialna za wykonanie zaimplementowanego zadania. Pojedynczy proces stanowi całkowity kontekst, który jest niezbędnym elementem do prawidłowego wykonania programu. Aby móc osiągnąć wysoką wydajność oraz jednocześnie w efektowny sposób wykorzystać posiadane zasoby sprzętowe koniecznie jest wykonanie pewnych fragmentów kodu w sposób równoległy, przez co zastosowanie mechanizmów wielowątkowych i wieloprocesowych jest kluczowym czynnikiem, który umożliwia tworzenie wysokowydajnych aplikacji. Jednak takie podejście do tworzenia aplikacji sprawiło, że powstał szereg nowych problemów związanych z komunikacją i synchronizacją powyższych mechanizmów. Bardzo ważne jest nie tylko zrozumienie samej problematyki aplikacji współbieżnych, ale również zrozumienie współbieżności osobno w kontekście wątku i procesu. Oba mechanizmy opierają się na niskopoziomowych funkcjach jakie są dostarczane przez system operacyjny. Pojęcie zasady ich działania jest koniecznym warunkiem, który umożliwia tworzenie wysokowydajnych i spójnych aplikacji. Sposoby pisania aplikacji przy wykorzystaniu obydwu mechanizmów bardzo się od siebie różnią nie tyko pod względem samego aspektu programowania ale również różnych przypadków zastosowania.

Na podstawie powyższych stwierdzeń można wywnioskować, że implementacja aplikacji współbieżnych jest o wiele trudniejsza w stosunku do aplikacji opierających się na pojedynczych wątkach i procesach, ale korzyści jakie dzięki temu można uzyskać są z pewnością wartę zachodu.

2. Przegląd literatury

W skład literatury, która pozwoliła dogłębnie przyjrzeć się problematyce artykułu wchodzą zarówno książki opisujące od podstaw działanie systemów operacyjnych [6], wątków i procesów oraz wszystkie inne tematy bezpośrednio z nimi związane, jak i publikacje naukowe pochodzące z serwisu Google Schoolar. Uzupełnieniem literatury jest zestaw prezentacji z wykładów „Systemy operacyjne” [4] oraz „Programowanie równoległe i rozproszone” [5] pochodzący z Politechniki Lubelskiej oraz szeroko dostępne artykuły w serwisie Wikipedia. Literatura jest również wspierana poprzez oficjalną dokumentację dla systemu operacyjnego i języków programowania [7] które zostały wykorzystane w celu przeprowadzenia badań.

Bardzo duża liczba pozycji z literatury skupia się opisie zasady działania wątków i procesów. Prezentują one sposób działania każdego z mechanizmów, który często jest poparty praktycznymi przykładami. Jednak w dzisiejszych czasach mimo bardzo rozwiniętej tematyki programowania współbieżnego w dalszym stopniu trudno znaleźć pozycję literaturową, która stanowiłaby porównanie tych dwóch mechanizmów jakimi są wątki i procesy.

3. Systemy operacyjne i współbieżność

W najprostszy sposób system operacyjny można zdefiniować jako środowisko, które przyjmuje działania wygenerowane przez użytkownika za pośrednictwem programów, które w dalszej kolejności są przetwarzane i zwracane z powrotem w postaci wyników. Do głównych zadań systemu operacyjnego należy zaliczyć:

- kontrolowanie pamięci operacyjnej dla działających zadań;
- obsługa sprzętu oraz urządzeń peryferyjnych;
- dostarczenie odpowiednich mechanizmów służących do prawidłowej komunikacji oraz synchronizacji między zadaniami;
- gromadzenie i zarządzanie danymi.

Współbieżność pozwala na to aby dany program był w stanie realizować kilka zadań jednocześnie w celu osiągnięcia jak największej wydajności. Opiera się ona na fakcie istnienia jednocześnie kilku wątków bądź procesów, które operują na współdzielonych zasobach. Z tego powodu ważnym warunkiem jest ich odpowiednia synchronizacja i komunikacja, ponieważ zarówno wątki jak i procesy są nawzajem od siebie zależne. Zastosowanie współbieżności wprowadziło charakterystyczne dla tego zagadnienia problemy, jednym z najprostszych i najbardziej rozpoznawalnych przykładów jest scenariusz konsumenta - producenta. W tym problemie istnieje jeden lub wiele procesów konsumenta i producenta, które razem współdzielą wspólną strukturę danych. Producent ma za cel wyprodukować zasób i umieścić go w strukturze tak, aby konsument mu go pobrać i skonsumować. Problemem jaki należy rozwiązać w tym zadaniu jest to, aby konsument nie pobierał danych w momencie kiedy nie ma ich w buforze, a producent nie nadpisywał nie odebranych danych w momencie kiedy bufor jest pełny.

Niepoprawna synchronizacja wątków i procesów może doprowadzić do takich problemów jak zakleszczenia (*ang. deadlock*) – czyli sytuacji w których dwa lub więcej procesów / wątków czeka na siebie nawzajem przez co żaden z nich nie jest w stanie się zakończyć, bądź zagłodzenia (*ang. resourcestarvation*) – czyli zjawiska w którym proces lub wątek nie jest w stanie ukończyć swojego zadania ponieważ nie może on uzyskać dostępu do procesora lub współdzielonego zasobu.

4. Procesy i wątki

Proces zdefiniowany jest jako pojedynczy egzemplarz wykonywanego programu ściśle związanego z zadaniem zleconym mu do wykonania. Rozpoczyna on swoje działanie w momencie wydzielenia mu przestrzeni adresowej oraz po przekazaniu procesorowi sygnału, aby ten zaczął interpretować odpowiedni kod. Każdy uruchomiony proces posiada swojego przodka (czyli proces z którego został on utworzony) i za pomocą funkcji systemowej *fork()* jest w stanie utworzyć proces potomny. Proces w momencie zakończenia działania zwraca do systemu operacyjnego odpowiednią wartość liczbową mówiąca o kodzie zakończenia procesu. Liczba 0 oznacza poprawne jego zakończenie, natomiast każda inna wartość stanowi odpowiedni kod błędu.

Wątek natomiast można zdefiniować jako fragment kodu, który wykonuje się równolegle w obrębie jednego procesu. Wątki są również mechanizmem dostarczanym i obsługiwanym przez system operacyjny, odstępstwem od tej zasady są wątki, które są zarządzane przez maszynę wirtualną w takich językach jak java lub python. Wątki podobnie jak procesy mają swój unikatowy identyfikator jednak w ich przypadku jest nim ciąg alfanumeryczny.

Wątki i procesy łączy i dzieli wiele podobieństw i różnic, do tych pierwszych można zaliczyć fakt, że wątek współdzieli większość zasobów (przestrzeń adresową, struktury danych systemowych itp.) jakie zostały przydzielone procesowi, wyjątkiem jest czas procesora, który jest przydzielany odrębnie każdemu wątkowi, warto również zaznaczyć to, że zakończenie działania wątku nie powoduje zwolnienia zasobów, ponieważ należą one do procesu. Natomiast do różnic między tymi dwoma mechanizmami można zaliczyć fakt, że wątki wymagają znacznie mniej zasobów i czasu do ich utworzenia niż procesy. Jednak jedną z najważniejszych różnic jaka dzieli wątki i procesy jest ich sposób komunikacji. Przesłanie pewnej ilości informacji między dwoma wątkami ogranicza się do odwołania do odpowiedniego wskaźnika danej struktury w której są przechowywane informacje, dodatkowo procesor zapewnia atomowość takich operacji. Natomiast przekazywanie danych pomiędzy dwoma procesami jest o wiele trudniejsze, wymusza to wykorzystanie mechanizmów komunikacji międzyprocesowej (*ang. Inter processcommunication*), które bazują na budowaniu dynamicznych struktur w pamięci systemów pozwalających na wymianę informacji pomiędzy procesami.

4.1. Biblioteka Boost

Biblioteka Boost w znacznym stopniu poszerza możliwości języka C++ poprzez zagregowanie szeregu różnych modułów zaczynając od ogólnego przeznaczenia kończąc na bibliotekach dedykowanych programowaniu współbieżnemu jaką jest Boost.Thread, która stanowi warstwę abstrakcji pozwalająca współbieżne wykonywanie zadań.

Wątek w bibliotece Boost jest reprezentowany przez klasę *boost::thread*, w celu utworzenia i uruchomienia nowego wątku należy wywołać następujący konstruktor *boost::threadnazwaWatku(&nazwaFunkcji, arg1, arg2, arg3, ...).* Pierwszym parametrem jest adres funkcji, która ma się wykonać w oddzielnym wątku, natomiast drugim opcjonalnym parametrem są argumenty, jakie są przekazywane do funkcji określonej pierwszym argumentem. W momencie gdy główny (wykonujący się w metodzie *main()*) wątek dojdzie do momentu utworzenia drugiego wątku poprzez jeden z wcześniej wymienionych sposobów, nowo utworzony wątek rozpocznie wykonywanie kodu funkcji przekazanej jako parametr. Kluczową funkcją jaką można znaleźć w klasie *thread* jest funkcja *thread::join()*

w momencie kiedy pierwszy wątek wywoła tą metodę na obiekcie klasy *thread,* reprezentujący drugi działający wątek, pierwszy wątek wstrzyma swoje działanie do momentu gdy praca drugiego wątku zostanie zakończona.

Muteks umożliwia wykonywanie sekcji krytycznej kodu, czyli takiego fragmentu, który powinien być wykonywany jednocześnie tylko i wyłącznie przez jeden wątek. Jest reprezentowany przez klasę *mutex.* Jego dwiema najważniejszymi metodami są metody *lock()* i *unlock().* Metoda *lock()* z nich blokuje wątek na obiekcie go wywołującym i wstrzymuje pozostałe wątki do momentu aż wątek obecnie wykonujący sekcję krytyczną wywoła metodę *unlock().*

Lock Guard wykorzystuje muteks w swojej implementacji (przekazywany jako argument konstruktora) i pozwala automatyczne zwolnienie blokady muteksu z momentem końca zakresu jego działania, dzięki temu podczas wystąpienia wyjątku muteks nie jest blokowany.

Unique Lock w przeciwieństwie do Lock Guard (który bezwarunkowo zakłada blokadę na dany zakres) pozwala na zablokowanie muteksu, sprawdzenie stanu i jego ewentualne odblokowanie podczas oczekiwania na spełnienie warunku, z tego powodu jest często wykorzystywany z zmienną warunkową.

Zmienna warunkowa jest mechanizmem, który został przeznaczony do sytuacji, w których jeden lub więcej wątków oczekuje na zajście jakiegoś warunku do momentu, kiedy inny wątek go spełni i wywoła odpowiednią metodę pozwalającą powiadomić i odblokować wcześniej zablokowane wątki. Połączone są one z metodami informowania o zmianie wartości zmiennej bez angażowania czasu procesora, który bez takiej informacji musiałby nieustannie sprawdzać czy warunek został spełniony lub nie. Zmienna warunkowa jest używana zawsze w parze z muteksem. W booscie zmienna warunkowa jest reprezentowana poprzez klasę *condition_variable*, posiada ona metodę *wait(),* do której przekazuje się *mutex,* umożliwia wstrzymanie wykonywania kodu przez dany wątek do momentu, aż na zmiennej tego obiektu nie zostanie wywołana metoda *notify_one()* lub *notify_all(),* która jest odpowiedzialna za wznowienie działania wcześniej wstrzymanych wątków.

Bariera [3] jest mechanizmem służącym do synchronizacji grupy wątków. Wątki po dojściu do bariery są na niej zatrzymywane do momentu aż ostatni z nich jej nie osiągnie, po tym wszystkie wątki są zwalniane i mogę kontynuować dalej swoją pracę.

Semafor jest to natomiast abstrakcyjny mechanizm, z którego korzystają zarówno wątki jak i procesy. Znajduje on zastosowanie podczas gdy wątki/procesy korzystają z wspólnego zasobu, dzięki czemu zapobiega on wykonaniu niechcianych operacji na określonym zbiorze danych. Semafor jest chronioną nieujemną liczbą całkowitą lub zmienną logiczną. W przypadku wykorzystania semafora w powiązaniu z danym zasobem, jego początkową wartość jest ustawiana na liczbę dostępnych zasobów tego typu. W chwili gdy wątek/proces chce skorzystać z danego zasobu, musi on sprawdzić czy wartość semafora jest dodatnia, co oznacza, że zasób jest dostępny. Natomiast gdy jego wartość wynosi zero oznacza to, że nie ma dostępnych zasobów i wątek/proces musi czekać aż inny tego typu mechanizm zwolni zasób i zwiększy wartość semafora. Na semaforze można wykonać dwie atomowe operacje:

- opuszczenie semafora (V) – zwiększa wartość zmiennej semaforowej, przez co współdzielony zasób staje się dostępny dla innych wątków/procesów;
- podnoszenie semafora (P) – jeśli wartość semafora wynosi zero, metoda blokuje wywołujący ją wątek/proces do momentu kiedy zostanie wywołana metoda opuszczenia semafora, po czym zmniejsza to wartość. Jeśli wartość jest większa od zera semafor nie blokuje wątku/procesu tylko zmniejsza jego wartość.

Biblioteka Boost sama nie dostarcza gotowej implementacji semafora, jednak dzięki niej można stworzyć własną implementację wykorzystując wcześniej omawiane mechanizmy synchronizacji.

4.2. Mechanizmy IPC

System operacyjny dostarcza wszystkie funkcje pozwalające na zarządzanie procesami [8], do najważniejszej z nich należy funkcja *fork()* służąca do utworzenia nowego procesu, który różni się od procesu macierzystego zazwyczaj tylko identyfikatorem PID. Kolejnymi istotnymi funkcjami służącymi do zarządzania procesami jest funkcja *getpid(),* która pozwala na pobranie identyfikatora PID obecnego procesu oraz funkcja *wait(),* która blokuje proces macierzysty ją wywołujący do momentu, aż proces potomny zostanie zakończony [2] przez co będą dostępne informacje o statusie jego zakończenia. System operacyjny udostępnia również szereg mechanizmów służących do przesyłania danych między procesami.

Potoki nienazwane [1] oraz nazwane są to dwa podstawowe mechanizmy służące do jednokierunkowego przesyłania i odbierania danych między procesami. Główną różnicą między nimi jest fakt, że potok nienazwany istnieje tak długo jak jest otwarty (do momentu zamknięcia wszystkich jego deskryptorów) natomiast potok nazwany istnieje jako plik w drzewie katalogów i jest identyfikowany przez jego nazwę. Funkcją służącą do utworzenia potoku nienazwanego (i jednoczesnego jego otwarcia) jest funkcja *pipe(),* która przyjmuje jako argument dwuelementową tablice liczb całkowitych i wypełnia ją w momencie wywołania kolejno deskryptorem potoku do odczytu oraz do zapisu. Aby wysłać dane do potoku należy posłużyć się funkcją *write()* natomiast, aby odczytać dane z potoku należy wywołać komplementarną metodę *read().* Do tworzenia potoków nazwanych służy natomiast metoda *mkfifo()*. Bardzo istotną różnicą między potokiem nienazwanym a potokiem nazwanym jest fakt, że deskryptor potoku nazwanego w momencie jego utworzenia nie jest otwierany w przeciwieństwie do potoku nienazwanego, przez co wymusza na programiście użycie wcześniej wspomnianej funkcji o nazwie *open()*.

Kolejki komunikatów w odróżnieniu od potoków są dwukierunkowe, co oznacza, że procesy komunikujące się ze sobą mogą przesyłać dane do siebie nawzajem, ponadto przesyłają one ustruktyryzowane porcje danych, gdzie potoki przesyłają dane w sposób niesformatowany, posiadają one również swój unikatowy identyfikator, który znajduje się

w systemowej tablicy zawierającej obiekty IPC. Proces nadający komunikaty może je wysłać nawet jeśli, żaden z innych procesów nie czeka na ich odbiór (w tym czasie każdy wysłany komunikat jest buforowany w kolejce). Są one przechowywane tak długo jak długo nie zostaną odebrane lub jak długo kolejka komunikatów nie zostanie usunięta. Proces odbierający komunikat ma możliwość odebrania pierwszego komunikatu dodanego do kolejki bądź komunikatu o określonym identyfikatorze. Do metod obsługujących kolejki komunikatów należą:

- msgget() - funkcja służąca do utworzenia nowej kolejki, bądź do pobrania już istniejącej;
- msgctl() - funkcja służąca do zarządzania kolejką;
- msgsnd() - służy do wysłania wiadomości do kolejki komunikatów;
- msgrcv() - pozwala na pobranie wiadomości z kolejki komunikatów.

Ogólna zasada działania semaforów została przybliżona w sekcji, która opisuje wątki. System UNIX pozwala tworzyć semafory jako zestaw składający się z pojedynczego semafora jak i kilku pojedynczych semaforów. Operacje na semaforach można wykonać poprzez zastosowanie poniższych funkcji:

- semget() - służy do utworzenia lub pobrania istniejącej tablicy semaforów;
- semop() - funkcja służąca do zmiany wartości semafora;
- semctl() - pozwala na wykonywanie takich operacji jak ustawianie początkowej wartości semafora, badanie praw dostępu do niego.

5. Metoda badawcza

Metoda badawcza opiera się na analizie teoretycznej wiedzy pozyskanej z literatury, która została poparta poprzez programistyczne rozwiązanie typowych problemów związanych z współbieżnością:

- Problem producenta – konsumenta;
- Problem wielu producentów – jednego konsumenta;
- Problem jednego producenta – wielu konsumentów;
- Problem czytelników i pisarzy;
- Problem pięciu ucztujących filozofów.

Każdy z powyższych problemów został rozwiązany osobno przy użyciu wątków i procesów, poprzez wykorzystanie odpowiednich funkcji udostępnianych przez powyższe mechanizmy, prezentuje to poniższa tabela:

Tabela 1. Mechanizmy wykorzystane przy rozwiązywaniu każdego z zadań.

Problem	Mechanizmy wykorzystane przy implementacji	
	Wątki	Procesy
Jeden konsument – Jeden producent	Mutex	Potok nienazwany
Wielu producentów- Jeden konsument	Mutex, Unique Lock, Conditional Variable	Potok nazwany
Jeden producent – Wielu konsumentów	Mutex, Lock Guard, Barriera	Potok nazwany
Czytelnicy i pisarze	Semafor (Muteks, ConditionalVariable)	Kolejka komunikatów
Pięciu filozofów	Semafor (Muteks, Conditional Variable), Barriera	Semafor

5.1. Przykładowy program

Przykładowy program będzie stanowił rozwiązanie klasycznego problemu konsumenta – producenta [9], w którym producent będzie produkował kolejne dane będące liczbami całkowitymi inkrementowanymi o jeden, po czym konsument będzie je odczytywał.

Przykład 1. Kod programu rozwiązujący problem konsumenta – producenta przy pomocy wątków.

```
boolwyprodukowanoWszystkieDane = false;
int dana = 0;
constintdanychDoWyprodukowania = 10;
mutexmutexWatku;

void watekProducent()
{
    cout<< "Tworzewatekproducenta: "
<<this_thread::get_id() <<endl;
while (dana <danychDoWyprodukowania)
   {
mutexWatku.lock();
dana++;
cout<< "Watekproducenta " <<this_thread::get_id() << "
wyprodukowaldane " << dana <<endl;
mutexWatku.unlock();
this_thread::sleep_for(chrono::milliseconds(5));
}
wyprodukowanoWszystkieDane = true;
cout<< "Koniec pracy producenta " <<this_thread::get_id()
<<endl;
}

void watekKonsument()
{
    cout<< "Tworzewatekkonsumenta: "
<<this_thread::get_id() <<endl;
   while (1)
   {
mutexWatku.lock();
cout<< "Watekkonsumenta " <<this_thread::get_id() << "
odczytujedane " << dana <<endl;
mutexWatku.unlock();
this_thread::sleep_for(chrono::milliseconds(5));
if (wyprodukowanoWszystkieDane)
      {
break;
      }
   }
cout<< "Koniec pracy konsumenta " <<this_thread::get_id()
<<endl;
}

int main()
{
   thread pierwszyWatek(watekProducent);
    this_thread::sleep_for(chrono::milliseconds(1));
threaddrugiWatek(watekKonsument);

pierwszyWatek.join();
drugiWatek.join();
exit(0);
}
```

Program posiada trzy funkcje *watekProducent(), watekKonsument()* oraz *main()*, która kolejno uruchamia dwie poprzednie metody w oddzielnych wątkach po czym czeka aż ich praca się zakończy. Do wykonania sekcji krytycznej kodu został wykorzystany muteks, który blokuje w wątku producenta moment, w którym zmienna *dana* jest inkrementowana, po czym zwalnia on blokadę, pozwalając

wątkowi konsumenta odczytać zainkrementowaną zmienną. Wątki działają do momentu osiągnięcia przez zmienną *dana* zadeklarowanej wartości 10.

Rozwiązanie tego samego problemu w przypadku procesów wygląda następująco:

Przykład 2. Kod programu rozwiązujący problem konsumenta – producenta przy pomocy procesów.

```
constintdanychDoWyprodukowania = 10;
int dana = 0;
intconstwielkoscBuforu = 50;

void procesProducent(intpdesk[2])
{
    char str[wielkoscBuforu];
while (dana <danychDoWyprodukowania)
   {
      dana++;
sprintf(str, "%d", dana);
printf("Producent %d produkuje dane: %s\n", getpid(), str);
write(pdesk[1], str, wielkoscBuforu);
if (dana == danychDoWyprodukowania)
{
        break;
      }
   }
}

void procesKonsument(intpdesk[2])
{
    char buf[wielkoscBuforu];
while (dana <danychDoWyprodukowania)
   {
read(pdesk[0], buf, wielkoscBuforu);
printf("Konsument %d konsumuje dane: %s\n", getpid(),
buf);
      dana = atoi(buf);
if(dana == danychDoWyprodukowania){
break;
      }
   }
}

main()
{
intpdesk[2];
   pipe(pdesk);

printf("Tworze proces producenta i konsumenta\n");

if (fork() == 0)
   {
procesProducent(pdesk);
printf("Procesproducenta %d konczydzialanie\n", getpid());
      exit(0);
   }

   if (fork() == 0)
{
procesKonsument(pdesk);
printf("Proces kosumenta %d konczydzialanie\n", getpid());
exit(0);
   }
wait(NULL);
printf("Procesy potomne procesu %d zakonczyly swoje
dzialanie\n", getpid());
exit(0);
}
```

Budowa struktury programu jest bardzo zbliżona jak w przypadku wątków, metoda *main()* tym razem tworzy dwa oddzielne procesy potomne (za pomocą metody *fork()*), po czym proces macierzysty czeka na zakończenie ich pracy poprzez wywołanie metody *wait*(). W celu zapewnienia synchronizacji i komunikacji pomiędzy procesami konsumenta i producenta został wykorzystany potok nienazwany. Proces producenta w momencie wyprodukowania danej zapisują ją w potoku za pomocą metody *write(),* jeżeli dana została już przesłana do potoku to proces konsumenta odczytuje ją za pomocą metody *read*(), jeżeli potok jest natomiast pusty to proces wstrzymuje swoje działanie na tej metodzie do momentu zapisania danych do potoku.

6. Wnioski

Wnioski zostały opracowane na podstawie zgromadzonych materiałów oraz wyników uzyskanych wskutek rozwiązania praktycznych problemów, pozwoliły one porównać wątki i procesy pod względem sposobów komunikacji i synchronizacji, stopnia trudności implementacji oraz wydajności.

Komunikacja pomiędzy procesami w stosunku do komunikacji pomiędzy wątkami jest w znacznym stopniu trudniejsza. Jak można zauważyć na praktycznym przykładzie, że przesłanie komunikatu pomiędzy wątkami ogranicza się do odwołania do odpowiedniej zmiennej, która dla każdego wątku w obrębie jednego procesu jest ulokowana pod tym samym adresem w pamięci komputera. Sytuacja wygląda całkowicie inaczej w przypadku procesów, ponieważ w momencie utworzenia nowego procesu (wywołaniu funkcji *fork()*), jest ona nowo lokowany w pamięci, przez co wymusza to korzystanie z wyżej poznanych mechanizmów komunikacji międzyprocesowej.

W przypadku procesów funkcje zapewniające synchronizacje są udostępniane przez system operacyjny i jednocześnie są funkcjami służącymi do komunikacji np. wywołanie metody *read()* na potoku wstrzymuje proces do momentu umieszczenia danych poprzez wywołanie metody *write()*. Mechanizmy synchronizacji dla wątków są dostarczane przez bibliotekę boost, która udostępnia szereg różnych funkcji pozwalających uzyskać różne zachowania wątków w zależności od potrzeb.

Istotnym kryterium porównawczym jest stopień trudności zrozumienia działania poszczególnych mechanizmów oraz ich implementacja. Należy się tutaj odwołać do różnic pomiędzy językiem C++ i C. Język C++ jest językiem obiektowym i to dla środowiska tego języka została stworzona biblioteka boost, dzięki wsparciu obiektowości pozwala on również podzielić aplikacje na poszczególne klasy, co sprawia, że kod staje się bardziej przejrzysty oraz w większym stopniu podatny na zmiany. Język C natomiast jest językiem strukturalnym, przez co stworzony kod może nie być tak dobrze czytelny w stosunku do kodu stworzonego w językach obiektowych.

Wydajność jest czwartym porównawczym czynnikiem, który przeważa na korzyść wątków. Wynika to z dużego kosztu utworzenia (lub zakończenia) procesu w stosunku do wątku, dzieje się tak dlatego ponieważ w momencie utworzenia nowego procesu, jest mu przydzielany całkowity nowy obszar pamięci, natomiast każdy nowo utworzony wątek współdzieli ten sam segment pamięci, który został przydzielony procesowi go tworzącemu.

Bazując na wyciągniętych wnioskach można stwierdzić, że wątki przeważają nad procesami pod różnymi aspektami, chociaż w praktyce jest to mocno uzależnione od przypadku zastosowania wyżej wymienionych mechanizmów.

Literatura

[1] Brzeziński Jerzy, Wawrzyniak Dariusz. Politechnika Poznańska. Materiały z zajęć Systemy Operacyjne. http://wazniak.mimuw.edu.pl/index.php?title=Systemy_operacyjne. 2006.

[2] Fusco John, Linux, Niezbędnik programisty. Helion 2009.

[3] Karbowski A., Niewiadomska-Szynkiewicz E., Programowanie równoległe i rozproszone. Oficyna Wydawnicza Politechniki Warszawskiej 2009

[4] Pańczyk Maciej, Politechnika Lubelska Materiały z przedmiotu Systemy Operacyjne 2014.

[5] Pańczyk Maciej, Politechnika Lubelska Materiały z przedmiotu Programowanie Równoległe i Rozproszone 2014.

[6] Love R., Linux, Programowanie systemowe. Helion. Gliwice 2008.

[7] Williams Anthony, BotetEscriba Vicente. Official documentation for Boost Library - Chapter 39. Thread. http://www.boost.org/doc/libs/1_68_0/doc/html/thread.html. 2007.

[8] Bershad, Brian N., et al. "User-level interprocess communication for shared memory multiprocessors." ACM Transactions on Computer Systems (TOCS) 9.2 (1991): 175-198.

[9] Guźlewski Z., Weiss T., Programowanie współbieżne i rozproszone w przykładach i zadaniach. WNT, Warszawa 1993.