ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1999
Seria: INFORMATYKA z. 37 Nr kol. 1421

Ryszard WINIARCZYK, Przemysław KOWALSKI , Jerzy KĘDZIERA
Instytut Informatyki Teoretycznej i Stosowanej, PAN

WYDAJNOŚĆ MECHANIZMÓW KOMUNIKACJI SYSTEMU QNX

Streszczenie. Praca prezentuje przeprowadzone testy wydajności komunikacji i synchronizacji dla systemu czasu rzeczywistego QNX. Przedstawiono wyniki, obrazujące wydajność komunikatów, potoków, przedstawicielstw, wykorzystania semaforów i sygnałów. Wydajność komunikatów i kolejek FIFO przedstawiono także dla pracy w sieci Ethernet. Dodatkowym elementem pracy są wyliczenia czasu przełączania procesów.

EFFICIENCY OF INTERPROCESS COMMUNICATION IN QNX

Summary. The article presents test for communication and synchronisation efficiency in the real-time operating system QNX. Results are presented illustrating the performance of the system for messages, pipes, proxies, semaphores and signals. Messages and FIFO queues were also tested in a local network based on Ethernet. The efficiency of process scheduling is evaluated.

1. Wstęp

1.1. Systemy czasu rzeczywistego

Komputerowe systemy czasu rzeczywistego są systemami, w których istotne są nie tylko wymagania funkcjonalne, lecz również ścisłe zależności czasowe: czas reakcji oraz maksymalny czas realizacji zadania. Systemy czasu rzeczywistego wyróżniają się wśród innych systemów następującymi cechami:

- realizacja zadania wymuszonego przez zdarzenie musi zostać zakończona przed upływem określonego czasu krytycznego reakcji;

- w razie pojawienia się błędu obliczeń lub przekroczenia czasu reakcji sygnalizowany jest błąd systemowy, który musi zostać obsłużony w zależności od aplikacji;
- dane z systemu czasu rzeczywistego są ważne tylko przez określony kwant czasu.

1.2. Testowanie systemów czasu rzeczywistego

Tworząc aplikację czasu rzeczywistego, użytkownik korzysta z oferowanych przez system operacyjny mechanizmów dla różnego typu usług. Istotna jest więc obiektywna ocena ilościowa parametrów mechanizmów systemu operacyjnego, działającego na danej platformie sprzętowej.

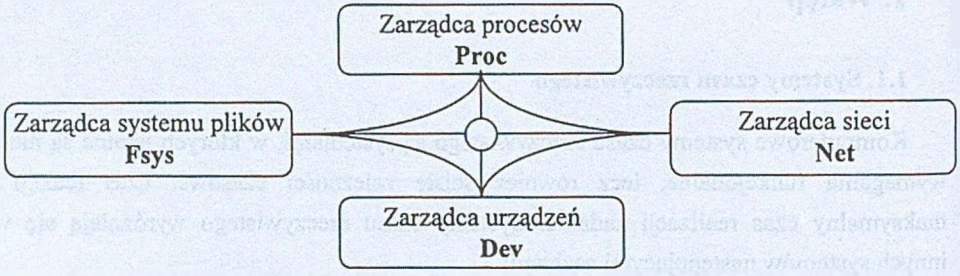
2. System operacyjny QNX

QNX jest sieciowym systemem operacyjnym czasu rzeczywistego działającym na komputerach opartych na architekturze iX86.

2.1. Architektura systemu

System operacyjny QNX posiada budowę modularną, co daje duże możliwości skalowania systemu. Jego architektura oparta jest na mikrojądrze oraz związanymi z nim procesami zarządców, które zasadniczo nie różnią się od zwykłych procesów. Każdy proces zarządcy realizuje określone zadania, korzystając z podstawowych usług jądra. Na przykład zlecenia usług realizowanych przez procesy zarządców są przekazywane za pośrednictwem przesyłanych komunikatów.

Rysunek 1. ilustruje konfigurację systemu QNX z wyszczególnionymi procesami zarządców, oraz ich komunikację z jądrem.

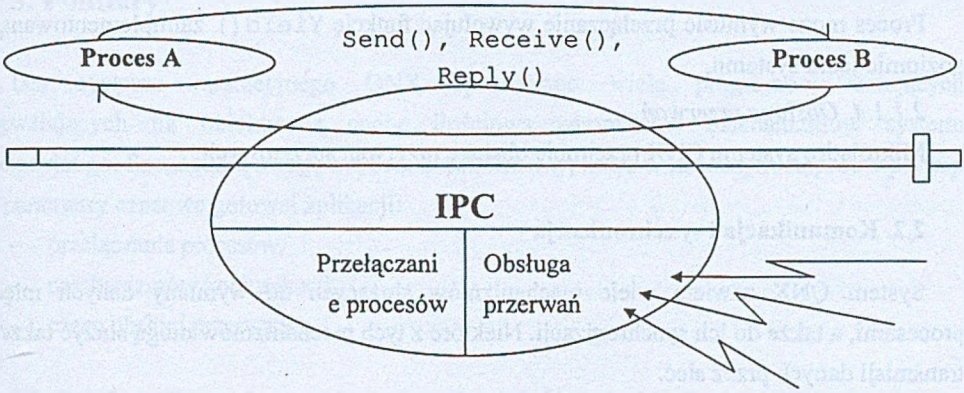


Rys. 1. Współpraca między procesami zarządców a mikrojądrem systemu QNX
Fig. 1. Co-operation between system managers and microkernel

2.1.1. Mikrojądro

W mikrojądrze systemu QNX zaimplementowano kilkanaście funkcji realizujących podstawowe zadania systemu, związanych z obsługą:

- lokalnej komunikacji międzyzadaniowej (IPC),
- przełączania procesów,
- przerwań.



Rys. 2. Funkcje mikrojądra
Fig. 2. Microkernel functions

2.1.1.1. IPC – Lokalna komunikacja międzyprocesowa

Jednym z podstawowych zadań mikrojądra systemu QNX jest zapewnienie komunikacji między procesami. Zaimplementowano na jego poziomie następujące metody komunikacji między procesami:

- Przekazywanie komunikatów – zapewniające zsynchronizowaną komunikację pomiędzy współpracującymi procesami (operacje: `Send()`, `(C)Receive()`, `Reply()`).
- Sygnały – służące przede wszystkim do zgłaszania sytuacji wyjątkowych. Jest to sposób znany z POSIX'a. Na poziomie jądra systemu zaimplementowano funkcję `Kill()` (wysłanie sygnału), oraz `Sigsuspend()` (oczekiwanie na nadejście sygnału).
- Przedstawicielstwa (ang. *Proxy*) – rozszerzające mechanizm sygnałów. Nie powodują blokowania procesu pobudzającego przedstawicielstwo, nie wymagają też potwierdzenia. W przeciwieństwie do sygnałów brak jest jednak możliwości pracy asynchronicznej. Pobudzenie przedstawiciela odbywa się za pomocą funkcji `Trigger()`, odbiór zaś wiadomości – tak jak i w przypadku przesyłu komunikatów – za pomocą funkcji `Receive()`.

2.1.1.2. Synchronizacja

Synchronizacji na poziomie jądra mogą posłużyć semafory (funkcja `Semafor`) oraz przesyłanie komunikatów.

2.1.1.3. Przełączanie procesów

Jądro odpowiedzialne jest za przełączanie procesów, aczkolwiek pełną obsługę zarządzania procesami zapewnia proces zarządzający `Proc`. W systemie QNX dysponujemy trzema algorytmami szeregowania procesów: FIFO, karuzelowym i adaptacyjnym.

Proces może wymusić przełączanie wywołując funkcję `Yield()` zaimplementowaną na poziomie jądra systemu.

2.1.1.4. Obsługa przerwań

Mikrojądro systemu QNX przejmuje obsługę przerwań sprzętowych.

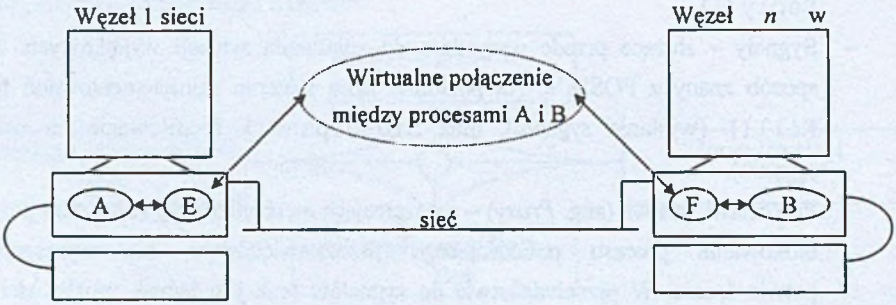
2.2. Komunikacja i synchronizacja

System QNX zawiera wiele mechanizmów służących do wymiany danych między procesami, a także do ich synchronizacji. Niektóre z tych mechanizmów mogą służyć także do transmisji danych przez sieć.

QNX w wersji 4 zapewnia usługi odpowiadające dwóm najniższym warstwom ISO/OSI. Połączenie pomiędzy dwoma procesami odbywa się za pomocą wirtualnego połączenia.

Łącząc w systemie QNX dwa procesy, uruchomione w różnych węzłach sieci, tworzy się wirtualne połączenie. Dla każdego z połączonych procesów jest ono widoczne jako proces. Dla przykładu, gdy łączymy proces A z węzła 1 i proces B z węzła *n*, utworzone zostanie wirtualne połączenie widoczne w węźle 1 jako proces E, a w węźle *n* jako proces F (rys. 3). Każde przesłanie komunikatu z procesu A do B w rzeczywistości będzie przesłaniem:

A → E → <poprzez sieć> → F → B



Rys. 3. Wirtualne połączenie
Fig. 3. Virtual circuit

gdzie E i F są nazywane procesami wirtualnymi. Wirtualne połączenie pozwala na przesył dwukierunkowy. W przypadku gdy jeden z procesów zostanie przerwany przed zakończeniem wirtualnego połączenia, zostanie ono również przerwane. Wirtualne połączenie przenosi komunikaty, oraz sygnały. Można również utworzyć zdalnych reprezentantów dla przedstawicielstw.

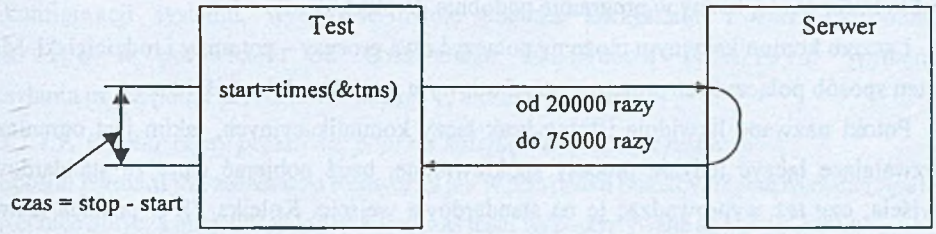
3. Pomiary

Dla systemu operacyjnego QNX opracowano wiele programów testujących, pozwalających na obiektywną ocenę ilościową parametrów mechanizmów systemu operacyjnego. Szczególną uwagę zwrócono na elementy, które w znacznym stopniu wpływają na parametry czasowe gotowej aplikacji:

- przełączania procesów,
- mechanizmów komunikacji,
- czasu obsługi przerwania.

3.1. Pomiary dla mechanizmów komunikacji i synchronizacji

Do pomiarów wydajności komunikacji między procesami czy to lokalnymi, czy to odległymi utworzono dwa procesy: testowy (test) i pomocniczy – dostarczający jednej usługi (zwracania danych do nadawcy) – serwer.



Rys. 4. Zasada realizacji testów
Fig. 4. The principle of software tests

Serwerem dla pomiarów wydajności przesyłu komunikatów, kolejek FIFO, wspólnej pamięci i semaforów, depozytów i przesyłania sygnałów był program *serwer*. Dla pomiaru wydajności przesyłu danych poprzez łącza komunikacyjne (potoki), ze względu na ograniczenia tego mechanizmu – program *test* utworzył serwer, poprzez podzielenie się przy użyciu funkcji `fork()`.

Ze względu na wymóg odesłania danych do nadawcy – zmierzony czas obejmuje zarówno czas przesłania danych, jak i obsługi przesyłu w obu procesach. Mierzony jest czas jednego przesyłu danych jako:

$$czas\ przesyłu = \frac{zmierzony\ czas - 2 * czas\ organizacji\ petli}{2 * liczba\ iteracji}$$

Taki sposób pomiaru może budzić wątpliwości w przypadku przesyłu komunikatów, które wymagają przesłań w obie strony (a zmierzony czas przesyłu reprezentuje jedynie czas przesłania danych w jedną stronę).

Liczba iteracji pętli, w której przesyłano dane, wahała się zwykle od 20000 do 75000 razy, podlegała też modyfikacjom (przed odpowiednie ustawienie parametrów programu), by dostosować się do wydajności komputera. Wynikało to z prostego faktu, że większa liczba iteracji zmniejszała błąd pomiaru, zwiększając czas testu – zwiększanie liczby iteracji nie powinno być nadużywane. W programie testowym istnieje możliwość zmiany liczby iteracji, aby móc dostosować test do szybkości testowego komputera.

Proces serwera rejestruje w systemie nazwę: `/test/serwer`, która umożliwia go, przez proces testujący, także w przypadku, gdy został zarejestrowany w innym węźle.

3.1.1. Potoki

Z systemu operacyjnego UNIX ([4]) QNX odziedziczył mechanizm potoków (ang. *pipe*).

Potoki dzielimy na nienazwane (inaczej łącza komunikacyjne) i nazwane (kolejki FIFO).

Potoki (obu rodzajów) służą do komunikacji jednokierunkowej. W potoku obowiązuje kolejność FIFO (ang. *First In First Out* – pierwszy wszedł, pierwszy wychodzi). Fizycznie jest to bufor obsługiwany w programie podobnie do pliku.

Łączem komunikacyjnym możemy połączyć dwa procesy – potomny i rodzicielski. Można w ten sposób połączyć też proces ze standardowym wejściem lub wyjściem.

Potoki nazwane likwidują istotny brak łączy komunikacyjnych, jakim jest ograniczenie pozwalające łączyć jedynie procesy spokrewnione, bądź pobierać dane ze standardowego wyjścia, czy też wyprowadzać je na standardowe wejście. Kolejka FIFO posiada nazwę w systemie plików – która umożliwia jej odnalezienie i podłączenie się dwóch różnych procesów, również niespokrewnionych.

3.1.1.1. Pomiar czasu przesłania danych przez potoki

Pomiary przeprowadzono w następujący sposób– proces testujący po inicjacji dzieli się na dwa procesy, które w pętli piszą i odczytują dane z potoków:

Rodzic:

```
while(i--) {
  write(pipe1[1], bufor, wPetli[seria]);
  read(pipe2[0], bufor, wPetli[seria]);
}
```

Dziecko:

```
while(i--) {
  read(pipe1[0], bufor, wPetli[seria]);
  write(pipe2[1], bufor, wPetli[seria]);
};
```

Znając czas wykonania owych dwu pętli z odczytem i zapisem do potoku, możemy obliczyć czas jednego przesłania.

Metoda ta różni się od przyjętej w opracowaniu [4], gdyż wymiana danych przez potoki, dla dwóch procesów, wymaga dodatkowego przełączania, wydłużając mierzony czas. Uzyskane w ten sposób wyniki były więc gorsze niż dla programu z opracowania [4] (dla programu testowego można w ciągu sekundy wykonać 3309,29 przesłań poprzez potok, dla programu testowego z pracy [4] czas wykonania 20000 przesłań wyniósł 5,12 sekundy – co daje 3906,25 przesłania na sekundę). Uzupełnieniem tego pomiaru był pomiar przesyłania danych poprzez potok w jednym procesie (bez przełączania).

Potoki są mało wydajnym mechanizmem. Dodatkową jego cechą jest silne uzależnienie od konfiguracji systemu, wydajność rośnie podczas korzystania z zarządcy potoków `/bin/Pipe` w porównaniu do stosowanego standardowo `/bin/Fsys`. Wydajność przesyłania przez potok przedstawia wykres 1, a czasy – wykres 2.

3.1.1.2. Pomiar czasu przesłania poprzez kolejkę FIFO (potok nazwany)

Sposób pomiaru jest zasadniczo identyczny jak w przypadku potoków nienazwanych (*pipe*).

Różnice dotyczą inicjalizacji kolejki. Podczas testu wykorzystywane są kolejki FIFO:

`//1/tmp/testfifo.1, //1/tmp/testfifo.2;`

`//1/ram/testfifo.1, //1/ram/testfifo.2.`

Jedynym mechanizmem mniej wydajnym od potoków (łączy komunikacyjnych) okazały się potoki nazwane. Potoków nazwanych nie wykorzystują zarządcy potoków, więc ich wydajność jest taka jak dla potoków nienazwanych wykorzystujących `/bin/Fsys`. Uzyskane wydajności przedstawia wykres 2, a czasy wykres 1. Wydajność przesyłu danych poprzez kolejkę FIFO była wyraźnie mniejsza niż dla komunikatów. Można w tym dostrzec efekt wykorzystania komunikatów jako pomocniczego mechanizmu (przesłanie danych poprzez kolejkę FIFO składa się z wysłania danych do zarządcy systemu plików `Fsys`, i z

Fsys do procesu – tak też jedno przesłanie poprzez potok musi trwać dłużej niż dwa przesłania poprzez komunikaty). Błąd pomiaru czasu dla kolejek nazwanych pracujących poprzez sieć był bardzo duży – stąd *okrągłe* czasy przesyłania danych (wykresy 3 i 4).

3.1.2. Przesyłanie komunikatów

Przesyłanie komunikatów jest podstawowym mechanizmem komunikacji dla systemu QNX, w którym, oprócz przesłania danych, mamy do czynienia z synchronizacją obu procesów.

Przebieg przesłania komunikatu jest bardzo prosty. Pierwszy proces wysyła komunikat – `Send()` – do drugiego procesu, po czym zostaje wstrzymany. Drugi proces odbiera komunikat – (C)`Receive()` – by następnie wysłać odpowiedź – `Reply()` – która przywraca działanie procesu pierwszego.

Przesłanie komunikatu odbywa się zawsze z potwierdzeniem. Długość komunikatu może wynosić 0 – takie przesłanie służy jedynie synchronizacji procesów.

3.1.2.1. Pomiar czasu przesłania komunikatu

Podstawowym mechanizmem komunikacji systemu QNX są komunikaty. Sposób pomiaru jest następujący: proces testowy wysyła `Send()` do procesu serwera, ten zaś odbiera dane (`Receive()`) i następnie je odsyła (`Reply()`). Każde `Send()` oznacza dwa przesłania:

`Send()` → `Receive()` – przesłanie program testowy → serwer

`Reply()` → `Send()` – przesłanie serwer → program testowy

Stąd też dzielimy uzyskany wynik przez dwa (otrzymując czas jednego przesłania). Korzystając z poniższych wyników, należy pamiętać, że proces odbiorcy (`Receive()/Reply()`) musi odpowiedzieć, gdyż nadawca czeka (w `Send()`) na odpowiedź – ogranicza to faktyczną wydajność dla komunikacji jednokierunkowej.

Przesył komunikatów jest najbardziej wydajnym sposobem wymiany danych pomiędzy procesami. Wydajność przedstawiona na rysunku 5 jest silnie uzależniona od czasu operacji na pamięci (wymiana danych między dwoma procesami na jednym komputerze, zasadniczo sprowadza się do kopiowania pamięci) – w testowanej konfiguracji można zaobserwować szczyt wydajności spowodowany optymalnym wykorzystaniem pamięci podręcznej – wydajność wzrastała do 28MB/sekundę. Po osiągnięciu tego apogeum wydajność spada (choć aż do prawie 8kB bloków danych jest większa niż dla operacji nie wykorzystujących pamięci podręcznej). Powyżej 8kB wydajność stopniowo rośnie, przekraczając 15MB/sekundę. Powyżej 8kB nie ma mowy o wykorzystaniu pamięci podręcznej w celu przyspieszenia operacji (przesyłana informacja nie mieści się w tejże pamięci). Dla komputerów z innymi procesorami niż 486DX, a także przy większym obciążeniu systemu maksymalna wydajność, jak i liczba bajtów w jednym przesłaniu, dla których ta maksymalna wydajność jest osiągana,

zmienią się. Nie zmieni się natomiast ogólna charakterystyka zależności wydajności do liczby przesyłanych bajtów. (Przykładowo dla komputera z procesorem PII 266MHz maksymalna wydajność została osiągnięta dla między 4 a 6kB; wynosiła ona ponad 300 mln. bajtów na sekundę.) Przeprowadzono też badania wydajności przy omijaniu wpływu pamięci podręcznej – rozwiązanie to polegało na zmienianiu adresu przesyłanego bufora przy każdym przesłaniu (przesunięcie w ramach dużego bufora) – złagodziło ono wpływ tejże pamięci. Skok wydajności ilustruje wpływ sprzętu na osiągane rezultaty.

Komunikaty także podczas pracy w sieci okazały się bardziej wydajne od kolejek FIFO, osiągając podczas testu około 80% maksymalnej teoretycznej wydajności sieci Ethernet.

3.1.3. Sygnały

Sygnały stanowią systemowy odpowiednik systemu przerwań, przejęty z systemu UNIX. Możemy oczekiwać na nadejście sygnału lub też połączyć nadejście sygnału z wywołaniem odpowiedniej procedury – co pozwala na asynchroniczną obsługę nadesłanych zgłoszeń.

Podstawowym zadaniem sygnałów jest przekazywanie komunikatów o błędach. Ze względu na przeznaczenie najczęstszym następstwem nadejścia sygnału jest awaryjne zakończenie procesu.

Dla użytkownika pozostawiono dwa sygnały, nie posiadające przypisanego im znaczenia. Mogą one posłużyć do synchronizacji procesów.

3.1.3.1. Pomiar czasu przesłania i obsługi sygnału

Obsługiwać sygnał możemy na dwu poziomach – na poziomie procesu (np. `sigsuspend()`):

Zawartość funkcji obsługi: `{};`

```
start = times(&tms);
while(licznik--) {
    kill(serwer,SIGUSR1);
    sigsuspend(&maska);
};
stop = times(&tms);
czasWykonania = stop - start;
```

A w serwerze: `sigsuspend(&maska);` `kill(klient,SIGUSR1);`

oraz na poziomie samej tylko funkcji obsługi (gdzie umieszczamy odpowiednią akcję):

Uwaga! zawartość funkcji obsługi: `kill(klient,SIGUSR2);`

(Serwer posiada identyczną pętle jak w pierwszym pomiarze, z wyjątkiem wykorzystania sygnału `SIGUSR2`). Poziom funkcji obsługi daje nam oczywiste plusy, jednak lista funkcji, które można wykorzystać wewnątrz jest dość ograniczona.

Sygnały są mniej wydajne od zbliżonego pod pewnymi względami mechanizmu przesyłu przedstawicielstw. Czas obsługi sygnałów przedstawiono w poniższej tabeli.

Tabela 1

Czasy obsługi sygnału

Sposób obsługi:	486DX2/66MHz	Pentium 75MHz
	Czas w [μs]	Czas w [μs]
Funkcja obsługi odsyła sygnał	47,7	28,3
Oczekiwanie na sygnał (`sisuspend()`);	69,7	38,0

W porównaniu z przedstawicielstwami na korzyść sygnałów przemawia możliwość asynchronicznego odbierania sygnałów (powiązania z sygnałem funkcji obsługi) – co może w praktyce uczynić je bardziej wydajnymi.

3.1.4. Przedstawicielstwa (ang. proxy)

Przedstawicielstwa stanowią pewną formę zarejestrowanych krótkich komunikatów (do 100 bajtów), ze sztywno określonym właścicielem. W przeciwieństwie do sygnałów, nie istnieje możliwość asynchronicznego wywoływania funkcji przez ich nadejście. Ze względu na dominującą w systemie rolę przesyłu komunikatów, odbierane w identyczny sposób przedstawicielstwa pozwalają uzupełnić listę typowych komunikatów zawierających zlecenia, o krótkie komunikaty, których wysłanie nie blokuje wysyłającego procesu. W porównaniu z sygnałami, użytkownik może utworzyć nie dwa, lecz wiele własnych przedstawicielstw, ich liczba ograniczona jest przez maksymalną liczbę uruchomionych procesów. Część literatury ([1]) określa je jako depozyty – nazwa ta jednak nie wydaje się zbyt szczęśliwa.

3.1.4.1. Pomiar czasu przesłania przedstawicielstwa

Tak jak i w pozostałych testach, na potrzeby pomiaru pracują dwa procesy – w pętli na zmianę wysyłają i odczytują przedstawicielstwa (program testowy: `Trigger()/Receive()`, program serwera: `Receive()/Trigger()`). Czas komunikacji za pomocą przedstawicielstw jest wyraźnie gorszy niż czas przesłania komunikatu, jeśli jednak przesyłamy krótkie, powtarzające się informację do jednego procesu docelowego i nie oczekujemy na odpowiedź, może się okazać mechanizmem najbardziej odpowiednim.

Przedstawicielstwa w porównaniu z sygnałami są szybsze. W porównaniu z przesyłem komunikatów (do którego są zbliżone) przedstawicielstwa są wolniejsze. Należy jednak pamiętać, że nie wymagają odsyłania odpowiedzi – będzie ona dodatkowym praktycznym opóźnieniem dla przesyłania komunikatów.

Przedstawicielstwa można wykorzystać do sygnalizowania nadejścia przerwań.

3.1.5. Pamięć dzielona

Wymianę danych między dwoma procesami można zrealizować generalnie na dwa sposoby – przesyłając komunikaty, bądź też współdzieląc fragment pamięci. W drugim z

przypadków, część pamięć jest dostępna dla kilku procesów, które mogą z niej w dowolnej chwili czytać lub pisać.

W systemie QNX istnieją dwa typy pamięci dzielonej – 16- i 32-bitowy. Pierwszy wywodzi się ze starszych 16-bitowych wersji systemu i tylko on może być stosowany przez programy 16-bitowe. Pamięć 32-bitową wprowadzono dla zgodności z POSIX'em – mogą jej używać tylko programy 32-bitowe. Obiekty typu pamięć 32-bitowa identyfikuje się za pomocą nazwy – podobnie jak pliki. W systemie QNX pamięć dzielona jest zlokalizowana w fizycznej pamięci komputera.

Niezsynchronizowany dostęp do pamięci dzielonej może być źródłem błędów i przekłamań – wymusza to poszukiwanie rozwiązań synchronizacji procesów.

3.1.6. Semafory

Ideę semafora zaproponował E. W. Dijkstra w 1968 roku –zdefiniowano wówczas ogólny semafor, na którym można wykonać dwie operacje ([6]):

- podniesienia semafora (często oznaczana jako V – sem_post). Dla semafora `S` jest to wznowienie jednego z procesów wstrzymanych przez opuszczenie tego semafora, lub jeśli takiego procesu brak `S:=S+1;`
- opuszczenie semafora (często oznaczane jako P – sem_wait). Dla semafora `S`, jeśli `S>0`, to `S:=S-1`, w przeciwnym razie wstrzymanie procesu wykonującego tę operację.

Te dwie operacje pozwalają na usystematyzowanie dostępu do współdzielonych zasobów.

3.1.6.1. Pomiar czasu obsługi sekcji krytycznej za pomocą semaforów

Przeprowadzono dwa pomiary – czasu przełączania dwóch procesów wymuszonego operacjami na semaforach (`sem_wait()` i `sem_post()`),

```
licznik = LP;
start = times(&tms);
while(licznik--) {
  sem_wait(semTest);
  sem_post(semSerwer);
}
stop = times(&tms);
czasWykonania = stop-start;
```

oraz czasu przesyłania w sekcji krytycznej danych za pośrednictwem pamięci wspólnej i funkcji `memcpy()`:

```
licznik = LP;
start = times(&tms);
while(licznik--) {
```

```
        sem_wait(semTest);
        memcpy( bufor, buf, wPetli[seria]);
        memcpy( buf, bufor, wPetli[seria]);
        sem_post(semSerwer);
    }
    stop = times(&tms);
    czasWykonania = stop-start;
```

Odpowiednik tej pętli znajduje się w programie pomocniczym (serwer) – zamienione są jedynie semafory – w miejscu `semTest` znajduje się `semSerwer` i odwrotnie. O ile pierwszy z pomiarów pozwolił ustalić czas wykonania operacji przełączenia wymuszonej operacjami na semaforach i oszacować liczbę możliwych przełączeń na sekundę, o tyle drugi pozwolił na dokonanie porównania wydajności przesyłania danych poprzez pamięć wspólną – tzn. jej wykorzystania z uwzględnieniem czasów zapisu i odczytu z pamięci.

Semafory umożliwiają szybką synchronizację procesów – przełączenie procesów wynikające z wykorzystania semaforów (czyli wykonanie pary `sem_wait()` (oraz wejście w stan `sem-blocked`) i `sem_post()` (oraz powrót ze stanu `sem-blocked`) zajęły 25,3μs, co dało 39 525,7 wejść do sekcji krytycznej na sekundę. Czas ten jest niezależny od rozmiaru sekcji krytycznej, ilości danych zapisywanych i odczytywanych. Z drugiej strony w praktyce po wejściu do sekcji krytycznej dokonuje się operacji zależnych od ilości danych – porównanie czasu przesyłania komunikatów z czasem kopiowania do pamięci wspólnej (`memcpy()`), zsynchronizowanych przez semafory wypadło na niekorzyść semaforów i `memcpy()` (patrz wykresy 1 i 2).

Zasadniczo wydajność semaforów jest jednak duża, należy też zwrócić uwagę, że powyższe porównanie przedstawiło wydajność semaforów i pamięci wspólnej dla zastosowania typowego dla przesyłu komunikatów – w wielu zastosowaniach semafory są niezastąpione.

W testach wykorzystywana jest pamięć dzielona, 32-bitowa: `//1/test/pamiec` – tylko z taką pamięcią prawidłowo funkcjonowały semafory.

3.1.7. Gniazda BSD dla TCP/IP

Wcześniej przedstawione mechanizmy komunikacji w systemie QNX ograniczały się jedynie do środowiska jednorodnego, choć mogły wspierać programy tworzone dla środowiska heterogenicznego. Dla systemie QNX można dokupić pakiet TCP/IP, który udostępnia mechanizm gniazd BSD, oryginalnie opracowanych dla systemu Unix. Również osobny pakiet przeznaczony jest dla programistów tworzących programy zawierające obsługę TPC/IP (TCP/IP Developer's Toolkit). Pakiet ten zawiera możliwość zdalnego wykonywania procedur (RPC).

Interfejs gniazd BSD jest obecnie bardzo rozbudowany. Pozwala na pracę zarówno z protokołem połączeniowym (TCP), jak i datagramowym (bezpołączeniowym – UDP).

3.2. Inne pomiary

3.2.1. Pomiar czasu przełączania procesów

Pomysł pomiaru jest bardzo prosty – mierzymy czas działania fragmentu programu bez wymuszonego przełączania procesów (`Yield()`), oraz czas działania prawie identycznego fragmentu, różniącego się jedynie pewną (znaną) liczbą przełączeń. Przy dużej liczbie (7,5 mln) przełączeń można dość dokładnie określić czas, jaki zajmuje jedno przełączenie (błąd pomiaru czasu jest niewielki). Program może wyglądać w sposób następujący:

```
/* Pętla bez przełączania*/
 licznik = LICZBAYIELD;
 start = times(&tms);
 while(licznik--);
 stop = times(&tms);
 czasBezYield = stop-start;

/* Pętla z przełączaniem*/
 licznik = LICZBAYIELD;
 start = times(&tms);
 while(licznik--) Yield();
 stop = times(&tms);
 czasZYield = stop-start;
 czasYield = czasZYield - czasBezYield;
```

Średni czas przełączania zależy od konfiguracji systemu i możliwości samego komputera. W testowej konfiguracji (486DX2 66MHz) czas ten wynosił około 8,2 μs.

3.2.2. Pomiar narzutu systemowego obsługi

Obliczono czas dodania funkcji obsługi dla istniejącego przerwania (0 – przerwanie zegara). Mała częstotliwość napływania przerwań wymusiła bardzo dużą liczbę iteracji, zorganizowanych w dwóch pętlach – zewnętrznej i wewnętrznej. Pętla bez przerwań ma więc postać:

```
licz2 = MAKS_ZEWN;
start = times(&tms);
while(licz2--){
  licz1 = MAKS_PRZERW;
  while(licz1--);
};
stop = times(&tms);
czasBezPrzerwan = stop-start;
```

Po pętli bez przerwań następuje podłączenie funkcji obsługi dla przerwania 0:

```
przer_licz=0;      /* Zerowany licznik przerwań */
if(id=qnx_hint_attach(PRZERWANIE,&przerwanie,FP_SEG(
  &przer_licz))) == -1) {
    printf("Nie moge dolaczyc przerwania!\n");
    return -1;
}
qnx_hint_mask(PRZERWANIE,1);
```

Dołączana funkcja obsługi przerwań ma postać:

```
#pragma off(check_stack);
pid_t far przerwanie(){
  ++przer_licz;
  return (0);
}
#pragma on(check_stack);
```

Sama pętla pomiarów ma postać identyczną jak pętla bez przerwań. Różnica czasu pomiędzy wykonaniem obu pętli, podzielona przez liczbę odebranych przerwań (`przer_licz`), daje w wyniku czas dołączenia funkcji obsługi do przerwania (czas wykonania samej funkcji jest pomijalnie mały).

Czas dodania funkcji obsługi wynosił około 3,5[μs].

3.3. Środowisko pracy

Można wyróżnić dwa środowiska pracy dla różnych typów pomiarów. Wynikało to z dostępnych instalacji systemu QNX, oraz ich konfiguracji. Pomiary wykonywano stosując system QNX w wersji 4.22B. Wszystkich pomiarów dokonano dla programów 32-bitowych.

3.3.1. Pomiary na pojedynczym komputerze

Dla badań na pojedynczym komputerze wykorzystano komputer z procesorem 486DX2/66MHz, wyposażony 8MB RAM. Pomiary wykonano zarówno dla systemu uruchomionego z twardego dysku, jak i uruchomionego z dyskietki. Błąd pojedynczego pomiaru czasu (`ticksize`) wynosi 10 milisekund.

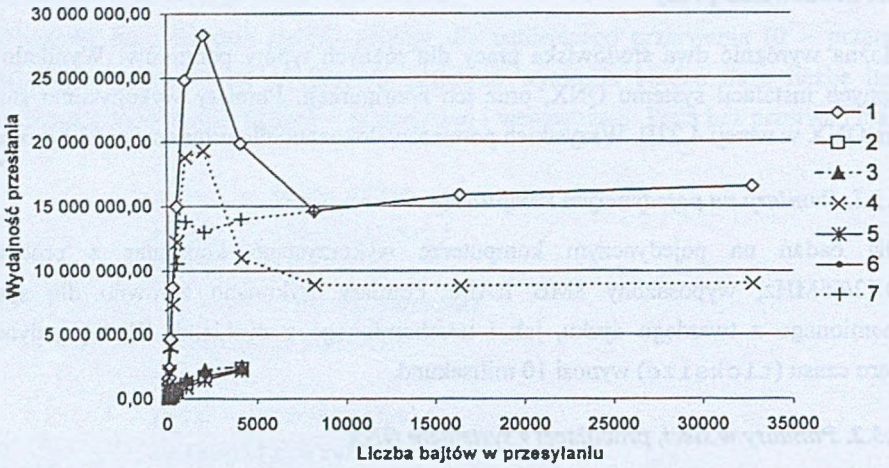
3.3.2. Pomiary w sieci, pracującej z systemem QNX

Pomiary dla sieci przeprowadzono wykorzystując dwa komputery połączone siecią Ethernet. Komputer, na którym uruchomiono program `serwer_test`, posiada procesor 486DX2/66MHz i 32MB pamięci RAM. Drugi komputer (na którym uruchomiono program `pr_test`) posiadał procesor 486DX taktowany zegarem 33MHz oraz również 32MB pamięci RAM. W testowanej sieci pracowało także kilka innych komputerów, co mogło wpłynąć na wyniki. Pomiary były mniej dokładne niż dla pojedynczego komputera – dłuższy czas przesyłania danych poprzez sieć wymusił zmniejszenie liczby iteracji.

4. Wydajność mechanizmów komunikacji – podsumowanie i porównanie wyników

Jeśli czas przesłania danych możemy określić jako

*czas przesłania = czas organizacji przesłania + liczba bajtów * czas przesłania bajtu*



Rys. 5. Wydajność dla konfiguracji jednostanowiskowej. 1 –komunikaty; 2 – potoki; 3 – potoki bez przełączania; 4 – przesyłanie z użyciem pamięci dzielonej i semaforów; 5 – kolejki FIFO; 6 – przedstawicielstwa;7 – przesył komunikatów bez użycia pamięci Cache

Fig. 5. Efficiency of communication on one computer. 1 – messages; 2 – pipes; 3 – pipes without switching; 4 – shared memory and semaphores; 5 – FIFO queues; 6 – proxies; 7 – messages without using CACHE memory

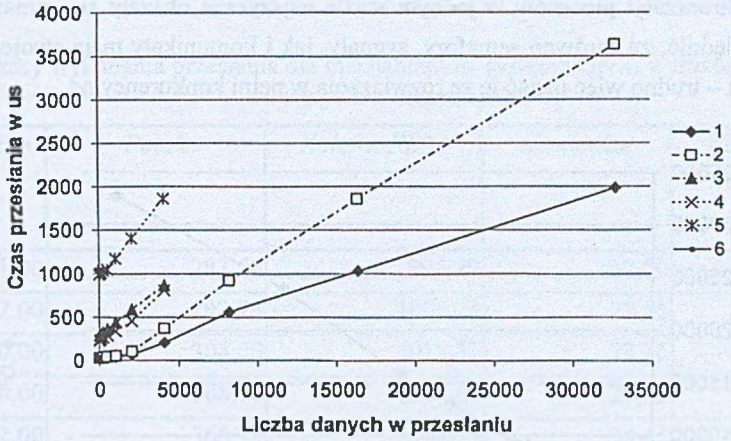
Tabela 2

Czasy pracy mechanizmów systemu QNX dla pracy lokalnej

Mechanizm komunikacji	Organizacja przesłania Czas w [μs]	Przesłanie jednego bajta Czas w [μs]
Łącza komunikacyjne	290	0,15
Łącze komunikacyjne bez przełączania	205	0,15
Kolejki FIFO	980	0,21
Komunikaty	27,3	0,06
Przedstawicielstwa	32,4	0,02
Semafory + memcpy ( ) *	33,5	0,11

Porównanie wydajności dla różnych mechanizmów dla pracy na jednym komputerze przedstawiają rysunki 5 i 6.

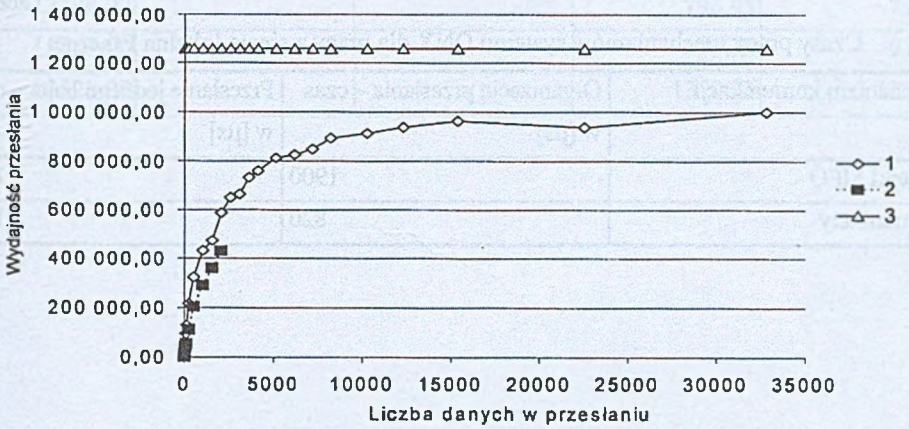
* Czas organizacji to prawie wyłącznie obsługa semaforów, czas przesłania jednego bajta wynika z czasu obsługi memcpy ( ) – w przypadku tego testu mamy dwukrotne kopiowanie z pamięci do pamięci, w przeciwieństwie do jednokrotnego dla przesyłania komunikatów.



Rys. 6. Czas przesyłania danych w konfiguracji jednostanowiskowej. 1 –komunikaty; 2 – przesyłanie z użyciem pamięci dzielonej i semaforów; 3– potoki; 4 – potoki bez przełączania; 5 – kolejki FIFO; 6 – przedstawicielstwa

Fig. 6. Communication times for one computer. 1 – messages; 2 – shared memory and semaphores; 3 – pipes; 4 – pipes without switching; 5 – FIFO queues; 6 – proxies

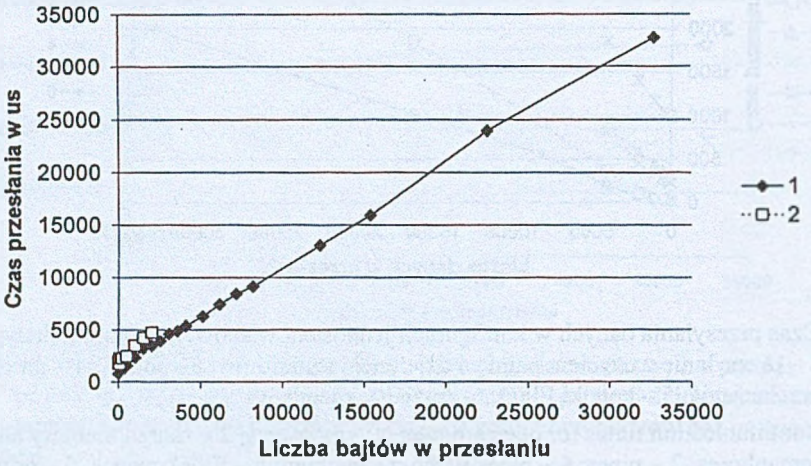
Dla przesyłania danych pomiędzy dwoma procesami na jednym komputerze najszybszym rozwiązaniem okazały się komunikaty, najwolniejsze zaś kolejki FIFO. Tylko wykorzystaniu programu `Pipe` potoki zawdzięczają pozycję lepszą niż kolejki FIFO. Pamięć wspólna wraz z semaforami okazała się nieco wolniejsza od komunikatów, należy jednak uwzględnić, iż ten sposób użycia pamięci wspólnej nie jest dla niej optymalny.



Rys. 7. Wydajność przesyłania danych między dwoma komputerami w sieci Ethernet: 1 – komunikaty; 2 –kolejka FIFO; 3 – maksymalna teoretyczna wydajność Ehternetu

Fig. 7. Efficiency of data transfer for two computers connected by Ethernet: 1 – messages; 2 – FIFO queues; 3 – maximal theoretical Ethernet efficiency

Dla synchronizacji procesów w jednym węźle najszybsze okazały się semafory. Należy jednak uwzględnić, że zarówno semafory, sygnały, jak i komunikaty mają swoje specyficzne zastosowania – trudno więc uznać je za rozwiązania w pełni konkurencyjne.



Rys. 8. Czas przesyłania danych w sieci Ethernet: 1 –komunikaty; 2 – kolejka FIFO
Fig. 8. Times for data transfer in Ethernet: 1 – messages; 2 – FIFO queues

Rezultaty osiągnięte dla pracy w sieci Ethernet przedstawiają rysunki 7 i 8. Można zauważyć o wiele niższą wydajność organizacji przesłania kolejek FIFO w porównaniu z komunikatami.

Tabela 3

Czasy pracy mechanizmów systemu QNX dla pracy z siecią lokalną Ethernet

Mechanizm komunikacji	Organizacja przesłania – czas w [μs]	Przesłanie jednego bajta – czas w [μs]
Kolejki FIFO	1900	1,45
Komunikaty	820	1,41

Tabela 4

Zawiera czasy wykonania przesłania dla mechanizmów systemu QNX, wyrażone w μs, dla testowych konfiguracji

Liczba bajtów	*Potok*	*Kolejka FIFO*	*Komunikat*	*Komunikat z omijaniem CACHE*
1,00	283,56	994,20	27,00	27,77
32,00	292,76	1000,28	27,55	29,67
100,00	304,00	1013,32	28,35	33,16
128,00	308,76	1017,80	29,20	34,10
512,00	368,40	1051,19	34,45	48,46
1024,00	441,96	1176,65	44,25	74,20
2048,00	591,96	1366,84	70,70	157,66
2560,00	658,76	1479,37	96,40	190,92
3072,00	727,96	1640,90	126,89	224,79
3584,00	797,12	1863,26	159,74	260,35
4096,00	869,16	2100,17	202,99	293,02
5120,00			312,63	360,31
6144,00			404,47	425,97
8192,00			558,91	561,06
10240,00			678,06	678,42
12288,00			796,95	797,78
15369,00			975,39	976,77
22528,00			1390,76	1391,27

Tabela 4 – c.d.

Liczba bajtów	*Przedstawicielstwo*	*Pamięć+ memcopy()*	*Potok w sieci*	*Komunikat w sieci*
1,00	32,10	30,75	1750,00	812,06
32,00	32,85	31,75	2000,00	833,28
100,00	34,30	33,35	2250,00	933,27
128,00		34,15	2250,00	973,33
512,00		44,69	2500,00	1566,67
1024,00		60,55	3500,00	2360,00
2048,00		109,65	4750,00	3480,00
2560,00		169,29		3933,33
3072,00		223,84		4619,33
3584,00		297,28		4880,00
4096,00		381,67		5379,33
5120,00		534,57		6292,67
6144,00		656,65		7426,67
8192,00		922,34		9152,67
10240,00		1192,02		42022,50
12288,00		1414,80		13085,33
15369,00		1747,88		15932,67
22528,00		2525,33		23978,00

Dla sieci Ethernet przyjęto maksymalną teoretyczną przepustowość sieci jako 1,2MB/s†. Przyjmując tę wartość uwzględniono maksymalną teoretyczną wydajność dla medium transmisyjnego (10Mb/s), efekt czasu propagacji i szybkości transmisji, wpływ minimalnego odstępu między ramkami, oraz wpływ nagłówka dla protokołu CSMA/CD. Maksymalna wydajność, możliwa do osiągnięcia, jest niższa – wpływ na nią mają także kontrolery sieciowe, oraz kolizje. Należy także pamiętać, że w testowanej sieci pracowały inne komputery, część z nich korzystała z sieci Internet (za pośrednictwem lokalnej sieci Ethernet) – co wpływało na osiągnięte wyniki.

† W artykule [5] przyjęto jako maksymalną wydajność sieci Ethernet 1,15MB/s.

LITERATURA

1. Sacha K.: QNX - system operacyjny, X-serwis, Warszawa 1995.
2. QNX 4 Operating System – System Architecture, QNX Software Systems Ltd. 1994.
3. WATCOM C Library Reference I & II, WATCOM International Corporation, Waterloo 1993.
4. Stevens W. R.: Programowanie zastosowań sieciowych w systemie UNIX, WNT, Warszawa 1995.
5. Abram M.: Nowości ze świata QNX, *Software 12/96 (24)* (na podstawie materiałów z QNXnews 2/96). Str. 70-71.
6. Weiss Z., Gruźlewski T.: Programowanie współbieżne i rozproszone w przykładach i zadaniach, WNT, Warszawa 1993.

Recenzent: Dr inż. Mirosław Skrzewski

Wpłynęło do Redakcji: 8 września 1998 r.

Abstract

Real-time operating systems have more rigorous resource management and scheduling than other systems.

One of real-time operating systems is a QNX. This operating system is flexible, because of its modular architecture. This architecture is based on a microkernel and system managers (Fig. 1.). The system managers have practically the same status as user-written programs. For an architecture like this, interprocess communication (IPC) is very important.

The fundamental form of IPC in QNX are messages. In QNX, a message is a packet of bytes which is synchronously transmitted from one process to another. They are very efficient (table 4 and fig. 5, 6, 7, 8).

A different, file-like form of inteprocess communication are pipes. There are two kinds of pipes: unnamed, and named (also called FIFO queues). They are easy to use, but inefficient.

A special form of messages are proxies. They are short (shorter than 100 bytes). The sending process does not need to interact with the recipient. They are especially suited for event notification.

Another IPC form are signals. It is a traditional form of IPC. They are used to support asynchronous interprocess communications.

Processes may co-operate not only by messages, but also by shared memory. There are two kind of shared memory: 16 bits memory from old versions of QNX, and new, POSIX shared 32 bits memory. We tested the second kind of memory, synchronised by semaphores.

QNX may work in a network. In the network, two forms of communications are used: FIFO queues and messages.

Efficiencies for this kind of QNX communications are presented in figure 5 for a local machine, and figure 7 for network. Figures 6 and 8, and table 4 present obtained times.

The article also presents times obtained for process scheduling.