Narzędzia programowania sieciowego

Mariusz Jarocki

14 września 2003

Wstęp

Skrypt „Narzędzia programowania sieciowego” przedstawia powszechnie używane techniki systemowe służące do realizacji *programowania współbieżnego* czyli takiego, w którym funkcjonalność programu komputerowego jest osiągana dzięki współpracy wielu równolegle wykonywanych fragmentów kodu oraz *programowania sieciowego*, w którym zadania cząstkowe delegowane są do programów umieszczonych na wielu maszynach w sieci komputerowej.

Skrypt ten powstał na potrzeby zajęć eksternistycznych z przedmiotu „Programowanie współbieżne i rozproszone”, realizowanego na Wydziale Matematyki Uniwersytetu Łódzkiego. Może być również materiałem pomocnicznym dla tradycyjnego trybu prowadzenia tego przedmiotu (również pod nazwą „Programowanie sieciowe”).

Autor zakłada, że uczestnik kursu spełnia następujące założenia:

- ma podstawową wiedzę z dziedziny systemów operacyjnych;
- potrafi programować w języku C na poziomie podręcznika B. Kernighana i D. Ritchie „Język ANSI C”;
- potrafi programować w języku Java na poziomie podręcznika K. Arnolda i J. Goslinga „Java”;
- zna podstawy pracy w systemie Unix;
- ma podstawową wiedzę o sieci Internet (TCP/IP).

Autor stara się nie zamieszczać w tym kursie informacji, które są wykorzystywane podczas realizacji przedmiotu „Algorytmy równoległe”, skupiając się na aspekcie technicznym programowania współbieżnego i rozproszonego. Stąd wynika różnica między tytułem skryptu a nazwą przedmiotu. Wdrażając programistę do narzędzi, powodujemy że realizacja przez niego zaawansowanych algorytmów

współbieżnych i rozproszonych będzie mogła skupić się na logice projektu. Będzie również potrafił dokonać właściwego wyboru narzędzi, tak aby nie napotkać z ich strony ograniczeń uniemożliwiających implementację w określonym zadaniu.

W materiałach dostarczonych studentowi znajdują się dokumenty niecytowane w niniejszej pracy, często dotyczące zagadnień tu nieomówionych. Zachęcam do lektury – materiał kursu charakteryzuje się bowiem dużą dynamiką wraz z wprowadzaniem nowych standardów i specyfikacji.

Kurs zawiera szereg zadań do samodzielnego rozwiązania. Większość z nich to programy lub modyfikacje programów. Od prowadzącego kurs zależy w jaki sposób zadania te będą wykorzystane. W intencji autora było wprowadzenie mechanizmu kontroli pracy studentów na podstawie dostarczanych przezeń rozwiązań kolejnych zadań.

Płyta CD z materiałami do kursu zawiera oprócz tego dokumentu również wybraną dokumentację dostępną w Internecie. Jej spis można znaleźć w pliku `tresc.txt` w katalogu głównym. Materiały do kursu są również dostępne w internecie pod adresem `http://www.math.uni.lodz.pl/~jarocki/nps/`, gdzie czytelnik znajdzie również odsyłacze do innych miejsc związanych z tą tematyką.

Środowisko systemowe Knoppix jest łatwo dostępne – obraz płyty może być na przykład ściągnięty z Internetu z wielu lokacji.

Rozdział 1

Podstawowe pojęcia systemu UNIX

1.1 Dlaczego UNIX?

Wybór środowiska systemowego dla realizacji materiału kursu wydaje się oczywisty – nowoczesne systemy unixowe (czy unixopodobne) zawierają wsparcie dla większości charakterystycznych form programowania współbieżnego i rozproszonego. Więcej – to rozwój tych właśnie systemów determinował standardy przemysłowe w tej dziedzinie. Systemy te są najlepiej dokumentowane, wiedza na ich temat jest najbardziej powszechna. Nie do przecenienia jest również fakt, że systemy te są łatwo dostępne (często darmowe) na wiele architektur komputerowych. System UNIX jest naturalną platformą takich środowisk programistycznych jak język C, C++, Java czy specyfikacji jak RPC czy CORBA.

Wymowny jest fakt, że w oryginalnych tytułach podręczników programowania sieciowego słowo UNIX jest często pomijane, jako że ta platforma wydaje się autorom w tym kontekście całkowicie oczywista.

Nie oznacza to jednak, że wiedza tu przedstawiona jest implementowalna jedynie na platformach unixowych. Standardy, wykreowane podczas rozwoju systemów unixowych, są wchłaniane do innych systemów. Możemy je znaleźć zarówno w nieunixowych systemach stacji roboczych, jak i w nieunixowych serwerach. Dodatkowo, techniki te legły u podstaw konstrukcji usuwającej heterogeniczność systemową platformy Java.

1.2 Znaczenie słowa UNIX

Czym tak naprawdę jest system Unix? Nazwa ta pojawia się po raz pierwszy w 1969 roku jako określenie systemu stworzonego przez Kena Thompsona i Dennisa Ritchie z zespołu badawczego Bell Laboratories, będącego częścią koncernu telekomunikacyjnego AT&T. Wraz z wersją 3 systemu pojawiła się przełomowa dla dalszych jego losów koncepcja rozwijania kodu systemu w języku C (powstałym dzięki Ritchiemu i Brianowi Kernighanowi), z niewielką nieprzenośną częścią pisaną w asemblerze danej maszyny. Od początku rozwój systemu był nakierowany na przenośność na poziomie kody źródłowego z racji używania w Bell Laboratories sprzętu rozmaitego typu. Wersja 7 z roku 1978 zamyka pierwszą fazę rozwoju systemu, ograniczoną do firmy AT&T, choć już w tym czasie kod systemu trafiał już do amerykańskich uniwersytetów. Unix stał się produktem firmy AT&T.

AT&T rozwijało dalej system, co zaowocowało opublikowaniem w 1982 wersji zwanej Systemem III, a w 1983 – Systemem V.

Na bazie kodu z Bell Laboratories były również prowadzone prace rozwojowe na uczelniach amerykańskich. Najbardziej wpływowy okazał się tu Uniwersytet Kalifornijski w Berkeley (UCB). Wersje systemu powstałe w UCB nazywane były BSD (Berkeley Software Distributions), a pierwsza kompletna pojawiła się już w roku 1978. Subsydia rządowe uczyniły wersje Unixa BSD równie dojrzałymi jak wersje z AT&T, choć rozwój szedł w nieco innych kierunkach. Zespół UCB zakończył prace badawcze wraz z wydaniem 4.4 z roku 1993.

Bazą każdego systemu Unix jest albo wydanie pochodzące z AT&T albo z UCB. Producent sprzętu (Sun, DEC, IBM, HP, Silicon Graphics) nabywał system Unix od firmy AT&T lub UCB, dostosowywał ją do własnego sprzętu i dodawał własne rozszerzenia tworząc wersje użytkowe systemu (odpowiednio: SunOS, Ultrix, AIX, HP/UX, IRIX).

Dookoła nazwy Unix powstało wiele organizacji standaryzujących. Sama nazwa Unix ma bogate dzieje, jeżeli chodzi o własność znaku towarowego. Sytuacje komplikuje dodatkowo fakt, że najpowszechniej obecnie wykorzystywana wersja systemu unixopodobnego, Linux, właściwie nie jest Unixem, ponieważ kod jądra został stworzony od podstaw przez Linusa Torvaldsa i wolontariat internetowy. Wolontariat rozwija również wersje systemu wywodzące sie z BSD jak FreeBSD, NetBSD oraz OpenBSD. Nad częścią systemów zbudowano środowiska użytkowe, często bardziej rozbudowane niż sama baza (Unixware nad Systemem V, Solaris nad SunOS–em, MacOS X nad FreeBSD). Definicja słowa Unix staje się powoli niemożliwa...

Może jest z Unixem tak jak z językiem SQL, który złośliwi definiują jako

dowolny język ze słowami kluczowymi SELECT, FROM i WHERE?

Bardzo ważnym pojęciem dla zrozumienia czym Unix jest a czym nie, jest norma POSIX (Portable Operating System Interface for uniX). Jest on rodziną standardów opracowanych przez IEEE (Institute for Electrical and Electronics Engineers), uznanych za normy przez ISO (International Organization for Standarization). Definiują one między innymi: interfejs między jądrem systemu (unixopodobnego) a językiem C, formaty baz systemowych, formaty archiwów, interpreter Bourne'a i inne programy usługowe (w tym m.in. *awk* i *vi*), komunikację międzyprocesową, rozszerzenia czasu rzeczywistego, interfejs wątków i inne. Popularnym znaczeniem słowa unixopodobny (ang. *unix-like*) jest „zgodny z normami POSIX". Pozwala to zakwalifikować jako systemy unixowe również te, które nie dziedziczą kodu po systemach z AT&T.

Zaproponuję więc inne spojrzenie na definicję słowa unix (celowo używam tu małej litery), spojrzenie oderwane od znaków towarowych, praw autorskich czy zapożyczeń kodu, spojrzenie traktujące unix jako zbiór charakterystycznych cech systemu (czyli słowo „unix" należy tu czytać jako „typowy system unixowy lub unixopodobny"), występujących najczęściej:

- Jądro systemu jest, oprócz pewnej części ściśle związanej z obsługiwanym sprzętem, napisane w języku wysokiego poziomu (C).
- Unix to system wielozadaniowy (z podziałem czasu); tradycyjnie unixy preferują zadania nastawione na wejście/wyjście, obniżając priorytety efektywne procesów obliczeniowych.
- Jednostką aktywną w systemie jest proces, pracujący w trybie nieuprzywilejowanym procesora, we własnej chronionej przestrzeni adresowej; jedynym elementem aktywnym w trybie uprzywilejowanym jest jądro systemu.
- Unix używa pamięci wirtualnej, rozszerzając pamięć operacyjną o tzw. *obszary wymiany* w pamięci masowej. Niewykorzystaną pamięć operacyjną wypełniają bufory używanych plików.
- Podstawową metodą tworzenia nowych procesów jest rozwidlanie procesu aktywnego funkcją systemową *fork*. Po jej wywołaniu system tworzy nowy proces, którego przestrzeń adresowa jest kopią przestrzeni procesu macierzystego. Oba procesy rozpoczynają współbieżną pracę od następnej instrukcji za wywołaniem *fork*.

  Często proces potomny wykonuje niedługo po utworzeniu funkcję systemową *execve*, która zastępuje kod aktywnego procesu kodem z pliku wykonywalnego.

- Procesy korzystają podczas pracy z mechanizmów *łączenia dynamicznego*, ładując kod wspólnych bibliotek w miarę potrzeb. Podstawową biblioteką uwspólnioną jest standardowa biblioteka języka C (tzw. *libc*).

- Komunikacja międzyprocesowa odbywa się przez jądro systemu (wydaje się, że jest to najmniej pożądana cecha systemu – większość środowisk unixowych zapewnia obecnie nie tylko wsparcie dla organizacji wspólnej pamięci, ale również elementów aktywnych w postaci wątków).

- Budowie interfejsu programisty systemu (API) przyświeca minimalizm, ujawniający się choćby tym, że odczyt i zapis informacji w rozmaitych urządzeniach obsługiwanych przez system odbywa się za pomocą tego samego interfejsu jak odczyt i zapis informacji do plików „zwykłych". Zasadę tą często definiuje się jako: „Dla unixa wszystko jest plikiem".

- Otwarty plik jest dostępny w procesie poprzez liczbę całkowitą zwaną *deskryptorem pliku*. Predefiniowanymi deskryptorami są tu wartości 0 (*standardowe wejście*, zwykle związane z klawiaturą terminala), 1 (*standardowe wyjście*, zwykle związane z wyjściem terminala) oraz 2 (*standardowe wyjście dla błędów*).

- Unix to system wielodostępny (o ile jego administator nie zażyczy sobie inaczej); typowymi formami realizacji wielodostępu są sesje z terminali tekstowych bądź ich emulatorów; dla fanów środowisk graficznych przygotowuje się zwykle zdalne środowisko pracy w oparciu o protokół X–Window.

- W środowisku tekstowym naturalnym środowiskiem pracy jest tzw. *interpreter poleceń* czyli *powłoka* (ang. *shell*).

- Unix wykorzystuje do pracy w środowisku rozproszonym rodzinę protokołów TCP/IP.

- Naturalnym sposobem organizacji pamięci masowej jest model indeksowy oparty na tzw. i-węzłach (ang. *i-nodes*). i-węzeł zawiera w postaci tablicy o stałym rozmiarze wszystkie informacje o pliku poza jego nazwą (jest to przyczyną problemów z implementacją w systemach unixowych tzw. list kontroli dostępu (ang. *access control lists*) – informacja taka z natury bowiem ma charakter dynamiczny). Odwzorowaniem i-węzłów na nazwy plików zajmują się pliki specjalne – katalogi.

- Unixowy system plików jest widoczny jako wielopoziomowe drzewo.

- Unix bezpośrednio po starcie widzi tylko jedno urządzenie pamięci masowej, zawierające tzw. korzeń systemu plików (oznaczany znakiem /). Inne

urządzenia są przyłączane do głównego drzewa w procesie tzw. montowania (dość nieszczęśliwa kalka angielskiego słowa *mount*) i są widoczne jako fragmenty drzewa plikowego od pewnego katalogu określanego jako punkt montowania.

- Plik danych jest ciągiem bajtów.
- System praw dostępu do plików (czyli również do urządzeń czy kanałów komunikacyjnych, w myśl hasła „wszystko jest plikiem") jest zbudowany w oparciu o tablicę bitową stałej długości, zapisaną w i-węźle. Zawiera ona zezwolenia na trzy podstawowe operacje – czytanie, zapis i wykonanie dla trzech rozłącznych klas użytkowników: właściciela pliku, członków tzw. grupy pliku oraz innych.

  Bardzo charakterystyczne dla unixa są bity SUID i SGID. Ich ustawienie dla pliku wykonywalnego zmienia efektywny identyfikator użytkownika (bądź włącza do grupy pliku) dla uruchamiającego.

  Unixowy system praw dostępu jest bardzo efektywny w działaniu, brak dynamicznych list dostępu jest jednakże dość uciążliwy.
- W unixie obowiązuje model administracjny, bazujący na ograniczonym zaufaniu do użytkowników. Ujawnia się on między innymi tym, że zwykle użytkownik lokalny ma prawo zapisu jedynie w swoim katalogu domowym, katalogu na pliki tymczasowe oraz w kilku innych, dobrze znanych miejscach. Jednocześnie administratora systemu (użytkownika o numerze identyfikacyjnym 0) nie dotyczą jakiekolwiek ograniczenia.

1.3 Podstawowa nomenklatura systemu UNIX i sieci Internet

Proces oznacza w Unixie: *wykonujący się program*. Każdy proces jest wyposażony w szereg cech takich jak:

- Unikalny identyfikator (liczba całkowita) zwany PID (*Process IDentifier*).
- Identyfikator procesu rodzica – PPID (od *Parent PID*).
- Identyfikator grupy procesów – PGID.
- Identyfikator użytkownika – UID (*User IDentifier*) oraz efektywny identyfikator użytkownika – EUID.

- Identyfikator grupy użytkowników – GID i analogicznie – EGID.
- Priorytet początkowy zwany liczbą NICE.

Proces pracuje w pewnym otoczeniu (środowisku, ang. *environment*). Ma ono postać dynamicznej tablicy asocjacyjnej łańcuchów znakowych: `nazwa=wartość`. Wartość otoczenia jest dziedziczona po procesie-rodzicu. Zmiany w otoczeniu procesu nie wpływają na otoczenia innych procesów, w tym na otoczenie rodzica.

Plik ma w systemach unixowych znaczenie szczególne – jak już powiedziano wcześniej, duża część funkcjonalności systemu jest realizowana za pomocą funkcji operujących na plikach.

Prawa dostępu do pliku są ustalane poprzez własności procesu – jeżeli na przykład proces należy do użytkownika X, a plik może być modyfikowany przez użytkownika X, to kod procesu może dokonać operacji modyfikującej plik.

Nazwy plików są hierarchiczne. *Korzeniem* (ang. *root*) wszystkich nazw jest katalog główny (/). Pełna nazwa ścieżkowa pliku zaczyna się od znaku /, składa się z zestawu nazw katalogów oddzielonych znakami / i kończy nazwą właściwego obiektu. Drzewo katalogowe jest najczęściej zbudowane funkcjonalnie tj. pliki określonego rodzaju są umieszczane w określonych miejscach i tak najczęściej zdarza się, że:

- lokacje /bin, /usr/bin i podobne zawierają publicznie dostępne programy wykonywalne; nazwy tych katalogów wchodzą w skład standardowej *ścieżki przeszukiwania dla poleceń* czyli zmiennej środowiskowej, która pozwala na używanie programów bez specyfikacji ścieżki do nich prowadzącej;
- lokacje /sbin, /usr/sbin itp. zawierają polecenia administracyjne;
- lokacje /lib, /usr/lib itp. składują biblioteki dzielone oraz biblioteki programistyczne;
- katalog /etc zawiera pliki konfiguracyjne systemu;
- katalog /dev zawiera pliki specjalne, pozwalające nam na kontakt z urządzeniami obsługiwanymi przez jądro systemu;
- w katalogu /var składuje się dzienniki pracy systemu, dane podsystemów kolejkujących oraz inne, szybkozmienne informacje;
- katalog /tmp jest przeznaczony na pliki tymczasowe.

Dostęp do wymienionych lokacji z pozycji nie-administratora jest zwykle ograniczony. Dla zapewnienia skutecznej pracy, każdy „żywy” użytkownik ma *katalog domowy* czyli miejsce na swoje prywatne dane.

Użytkownik „żywy” wchodzi w interaktywny kontakt z systemem najczęściej za pomocą tzw. *powłoki* (ang. *shell*) czyli interpretera poleceń (inne formy interakcji są obecnie w ofensywie).

Każde urządzenie w sieci Internet (IPv4), która będzie tu podstawowym środkiem komunikacji rozproszonej ma conajmniej jeden identyfikujący ją *adres IP*, składający się z czterech bajtów. Zwyczajowo adres ten zapisuje się w postaci czterech liczb dziesiętnych, oddzielonych kropkami. Aby ułatwić korzystanie z urządzeń sieciowych, zwykle z adresem IP łączy się jego nazwa symboliczna w postaci łańcucha znaków ASCII, dostarczana przez różne usługi nazewnicze.

W naszej pracy będziemy korzystali głównie ze specjalnego adresu IP, 127.0.0.1 (nazwa: `localhost`) oznaczającą nasz komputer.

Każda *usługa transportowa* w sieci Internet jest kojarzona przez 16–bitową liczbę całkowitą zwaną *numerem portu*. Port jest „przywiązywany” do procesu, stąd też pełny adres punktu międzyprocesowego transportu sieciowego jest identyfikowalny przez 6 bajtów: 4 bajty adresu IP i 2 bajty numeru portu. Między punktami transportu korzystamy zwykle albo ze strumieniowego sposobu przesyłania (protokół TCP), albo przesyłamy paczki danych zwane *datagramami* (protokół UDP).

Rozdział 2

Model pracy w trakcie kursu

2.1 Podstawy pracy w środowisku Knoppix

Z kursem dostarczona jest płyta CD zawierająca środowisko Knoppix – uruchamialną z płyty wersją systemu Linux wraz z bogatym zbiorem oprogramowania użytkowego, w tym ze wszystkimi potrzebnymi podczas kursu narzędziami programistycznymi. Informacje o tym środowisku można uzyskać m.in. z serwisów internetowych `http://www.knoppix.de` oraz `http://knoppix.7thguard.net` (po polsku). Środowisko jest przystosowane do działania na większości platform PC dostępnych na rynku.

System Knoppix nie zostawia domyślnie widocznych śladów w strukturze pamięci masowej komputera. Zaletą tego rozwiązania jest to, że nie musimy go instalować na naszym komputerze, wadą natomiast to, że musimy sami zadbać o to, aby nasze dane, które wytworzymy podczas kursu były zapisane w miejscu trwałym. Możemy użyć w tym celu na przykład twardego dysku naszego komputera, którego odpowiednie partycje możemy przyłączyć do drzewa Knoppixa. Mogą to być partycje sformatowane w systemie DOS czy Windows (ale raczej nie partycje typu NTFS, przygotowane przez system Windows NT i sukcesorów). W takim przypadku lepiej użyć dyskietki.

Bezpośrednio po załadowaniu rekordu ładującego możemy dostosować start Knoppixa do konkretnej maszyny. Środowisko można uruchomić w trybie graficznym (zalecana pamięć operacyjna 96 MB) poprzez naciśnięcie Enter bezpośrednio po znaku zachęty `boot:` lub tekstowym przez wprowadzenie tekstu `knoppix 2`. Inne opcje ładowania możemy ujawnić przyciskiem F2.

W trybie tekstowym system startuje z czterema konsolami tekstowymi z uruchomionym interpreterem poleceń, pracującym z prawami administratora (mo-

żemy to poznać po znaku # kończącym tekst zachęty oraz po słowie `root` go zaczynającym). Przełączenie między konsolami odbywa się za pomocą kombinacji klawiszy Alt+F1, Alt+F2 itd. Wiele konsol bardzo się przyda w pracy z kursem, choćby po to, aby na jednej uruchomić serwer komunikacyjny, na drugiej – jego klienta, a z trzeciej kontrolować ich zachowanie w systemie.

Wydaje się, że dobrym pomysłem jest pozbycie się na wstępie pracy na konsoli tekstowej praw administratora. Wykonujemy to poleceniem:

```
su - knoppix
```

`knoppix` jest tu użytkownikiem, z którego prawami pracuje m.in. konsola graficzna i który ma pewne dodatkowe prawa związane z obsługą systemów plików. Pozbycie się praw powinniśmy zaobserwować w tekście zachęty. Polecenie `exit` spowoduje powrót do pracy w trybie administratora.

Jeżeli zdecydujemy się na pracę w konsoli graficznej, pamiętajmy o kombinacjach Ctrl+Alt+F1, Ctrl+Alt+F2 itd., za pomocą których możemy przełączyć się do odpowiednich konsol tekstowych. Powrót do konsoli graficznej zapewnia kombinacja Alt+F5.

Konsola graficzna jest domyślnie skonfigurowana do działania ze popularnym środowiskiem KDE o intuicyjnej obsłudze. Ponieważ nasza praca będzie wykonywana za pośrednictwem interpretera poleceń, powinniśmy uruchomić tu program konsoli tekstowej (z dolnego paska narzędzi, poprzez ikonę z monitorem i muszelką). Podczas pracy w tym programie, możemy tworzyć (poprzez przycisk „New”) i zamykać (poprzez polecenie `exit` wydane z interpretera) wiele emulatorów terminala tekstowego.

Przyłączenie istniejącej partycji dyskowej (przyjmujemy, że użytkownik ma komputer wyposażony w jeden dysk IDE, przyłączony do tzw. pierwszego kanału w trybie Master; jest to typowa konfiguracja) jest osiągalne dzięki poleceniu:

```
mount /mnt/hda1
```

Liczba 1 może być zastąpiona liczbą 5, 6 itd. o ile użytkownik będzie chciał wykorzystać dalsze partycje swojego dysku, przygotowane przez system DOS czy Windows. Partycja będzie widoczna jako zawartość katalogu /mnt/hda1. Najwygodniej będzie poleceniem `cd /mnt/hda1` zmienić teraz katalog bieżący.

W przypadku niemożności pracy z dyskiem twardym, zaleca się korzystanie z katalogu domowego użytkownika `knoppix` (jest on umieszczony na ramdysku), a przed zakończeniem pracy przekopiowanie jej rezultatów na dyskietkę poleceniem:

```
cp jakis_plik /mnt/floppy
```

Przywrócenia danych z dyskietki z powrotem do katalogu domowego użytkownika można dokonać poleceniem:

```
cp /mnt/floppy/jakis_plik .
```

Kropka oznacza tu katalog bieżący. Zamiast nazwy pliku można użyć symboli globalnych (* oznacza dowolny ciąg znaków).

Zaleca się przegranie (za pomocą własnego systemu operacyjnego) plików z płyty „Materiały” (katalog kurs) na dysk twardy lub dyskietkę przed rozpoczęciem właściwego kursu.

Zamknięcie systemu Knoppix dokonujemy za pomocą polecenia `halt` wydanego z poziomu administratora z konsoli tekstowej lub poprzez zamknięcie środowiska graficznego (analogicznie jak w systemie Windows).

Jako edytora tekstu możemy użyć programów: `mcedit` (edytor programu Midnight Commander, klonu znanego powszechnie programu Norton Commander), `jed`, `nano` czy `vi`. Uruchamiamy je z nazwą pliku źródłowego w argumencie. Można też wykorzystać edytory oferowane w środowisku KDE.

Polecenie `man tekst` wyświetla tekst z elektronicznego podręcznika systemu, poświęcony poleceniu czy funkcji `tekst`. Wyjście z trybu przeglądania podręcznika zapewnia klawisz „q”.

2.2 Diagnozowanie działania programów

Do diagnozowania działania programów z kursu przydatnych będzie kilka prostych poleceń.

Polecenie `ps` wyświetla listę procesów, które należą do nas i są przywiązane do tego samego terminala co interpreter poleceń z którego zostało wydane. `ps x` wyświetla wszystkie nasze procesy. `ps aux` (lub `ps -ef`) wyświetla wszystkie procesy, również nienależące do nas.

Polecenie `pstree` wyświetla listę procesów w postaci drzewa. Z opcją `-p` dostajemy oprócz nazw również PID–y procesów, z opcją `-u` – właścicieli. Wariant `pstree knoppix` pozwala na ujawnienie tylko procesów należących do użytkownika `knoppix`.

Polecenie `top` pozwala nam na śledzenie procesów systemu w sposób ciągły. Klawisz „h” wyświetla pomoc do tego programu.

Wątki są raportowane tak jak procesy (PID procesu/wątku nie jest równy identyfikatorowi wątku!).

Polecenie `kill -XXX PID_procesu` wysyła do procesu sygnał SIGXXX np. `kill -INT 390` wysyła sygnał SIGINT do procesu, którego PID wynosi 390. Polecenie `killall -XXX nazwa_procesu` wysyła odpowiedni sygnał do wszystkich procesów o podanej nazwie.

Polecenie `nc localhost nr_portu` nawiązuje połączenie TCP z lokalnym komputerem na podanym porcie. Dane wysyłane na port będą pojawiały się na konsoli, dane wpisywane na standardowe wejście będą transmitowane w drugą stronę, aż do zerwania połączenia lub naciśnięcia Ctrl+C (podobnie działać będzie polecenie `telnet localhost nr_portu`; jego przerwanie osiągamy przez kombinację Ctrl+] i „q” Enter). Polecenie `nc -l -p nr_portu` instaluje na wybranym porcie tzw. „serwer echa”.

2.3 Kompilacja i uruchamianie programów

Kompilacji programu w języku C dokonujemy poleceniem:

```
cc nazwa_pliku_zrodlowego.c
```

Kompilator tworzy program wykonywalny w pliku o nazwie `a.out`. Jeżeli chcemy sami nadać nazwę plikowi wykonywalnemu używamy konstrukcji:

```
cc nazwa_pliku_zrodlowego.c -o nazwa_pliku_docelowego
```

Zwyczajowo nie nadajemy rozszerzeń programom wykonywalnym, więc powyższe często ma postać:

```
cc prog.c -o prog
```

Programy wieloplikowe kompilujemy wymieniając wszystkie pliki źródłowe. Jeżeli zachodzi konieczność dołączenia pozastandardowej biblioteki wsparcia programistycznego (a tak dzieje się np. przy korzystaniu z biblioteki wątków posiksowych) używamy formuły:

```
cc nazwa_pliku_zrodlowego.c -o nazwa_pliku_docelowego
  -lnazwa_biblioteki
```

co oznacza dołączenie do kodu biblioteki z pliku `[/usr]/lib/libnazwa_biblioteki.a`.

Udana kompilacja jest „cicha”.

Uruchamianie programów znajdujących się poza ścieżką przeszukiwania wymaga podania choćby najkrótszej ścieżki do nich prowadzącej. Przy plikach wykonywalnych z katalogu bieżącego, uruchamianych bez argumentów może to wyglądać następująco:

```
./nazwa_programu
```

Kompilacja programu–klasy w języku Java odbywa się za pomocą polecenia:

```
javac NazwaPliku.java
```

a jej rezultatem – jeden lub wiele (w zależności od tego ile definicji klas znajdowało się w pliku źródłowym plików z przyrostkiem `.class`. Uruchomienia statycznej funkcji głównej z klasy dokonujemy przez:

```
java NazwaKlasy
```

Uwaga – nie podajemy w tym przypadku rozszerzenia (`.class`).

Użytkownicy systemów typu DOS czy Windows i/lub pascalopodobnych języków programowania powinni uważać na wielkość liter. Zarówno system nazewniczy plików w Unixie, jak i języki programowania C i Java odróżniają małe i wielkie litery.

Rozdział 3

Osiąganie współbieżności

3.1 Sygnały

Sygnałem nazywamy informację w postaci liczby całkowitej przesyłanej przez jądro systemu do procesu. Za pomocą odpowiedniej funkcji (*kill* z signal.h) proces może przesłać sygnał do procesu będącego we władaniu tego samego użytkownika (ograniczenie to nie dotyczy użytkownika o numerze 0), w tym do samego siebie.

Sygnał dociera do odbiorcy w sposób asynchroniczny tj. proces nie czeka na jego wystąpienie. Może on być przez odbiorcę zignorowany, obsłużony przez standardową procedurę lub obsłużony przez procedurę zdefiniowaną przez użytkownika. Obsługa sygnału wiąże się z przerwaniem wykonywania „normalnej" nitki sterowania, wykonanie zdefiniowanego fragmentu kodu oraz powrót do „normalnego" działania. W czasie realizacji procedury obsługi sygnału, inne sygnały tego samego typu są blokowane. Blokowanie rozumiane jest tu w ten sposób, że sygnał zostanie powtórnie wygenerowany, choć tylko raz mimo że zablokowań tego typu sygnału mogło być wiele.

Przykład 1 (signal1.c) *Program ten obsługuje sygnał SIGINT, który jest generowany przez interpreter poleceń do pierwszoplanowego zadania po naciśnięciu kombinacji Ctrl+C. Instalacji procedury obsługi sygnału dokonujemy w wywołaniu funkcji* signal. *Jej drugi argument jest nazwą funkcji o nagłówku analogicznym do użytej tu* handler_sigint. *Program generuje co sekundę znak kropki na standardowe wyjście. Naciśnięcie kombinacji Ctrl+C nie przerywa procesu, jak to dzieje się standardowo lecz powoduje wypisanie stosownego komunikatu. Zatrzymanie jest w dalszym ciągu możliwe przez np. kombinację Ctrl+\ (Ctrl+Backslash).*

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler_sigint(int sig)
{
    printf("\nNaciśnięto Ctrl+C\n");
}

main()
{
    signal(SIGINT, handler_sigint);
    for (;;) {
        printf(".");
        fflush(stdout);
        sleep(1);
    }
}
```

Dużo problemów stwarza dotarcie sygnału do procesu który zablokował się w funkcji jądrowej (np. czeka na dane ze standardowego wejścia). Zwykle funkcje te nie są wznawiane i sygnalizowany jest (w zmiennej *errno*) błąd EINTR (stała z errno.h).

Zadanie 1 *W poprzednim przykładzie usuń z kodu generowanie w procedurze obsługi sygnału wywołanie funkcji* printf *(procedura sygnału będzie miała puste ciało). Dlaczego naciśnięcie Ctrl+C „przyspiesza" generowanie kropek?*

3.2 Rozwidlanie

Wywołanie w programie funkcji bibliotecznej *fork* (unistd.h) powoduje utworzenie procesu, który rozpoczyna współbieżną pracę z procesem który fork wywołał. Przestrzeń adresowa nowego procesu jest dokładną kopią przestrzeni adresowej rodzica z chwili wywołania fork (z jedną różnicą). Zarówno proces macierzysty jak i nowo utworzony wykonuje jako pierwszą instrukcję występującą w kodzie tuż po wywołaniu fork.

Przykład 2 (fork1.c) *Program dokonujący rozwidlenia, po którym oba procesy wykonują pętle nieskończone (przerwij wykonanie przez Ctrl+C)*

```
#include <stdio.h>
#include <unistd.h>

void loop(char c)
{
    for (;;) {
        printf("%c", c);
        fflush(stdout);
        sleep(1);
    }
}

main()
{
    fork();
    loop('*');
}
```

Funkcja loop generuje co sekundę znak gwiazdki na standardowe wyjście. W funkcji głównej proces rozwidla się i zarówno proces rodzicielski jak i potomny zapętlają się w funkcji loop.

Zadanie 2 *Spróbuj przewidzieć częstotliwość generowania znaku * jeżeli wywołanie funkcji fork w poprzednim programie występowałoby dwa razy.*

Podstawową rzeczą dla opanowania rozwidlenia jest zróżnicowanie kodu wykonywanego przez proces rodzicielski i potomny. Przychodzi nam tu z pomocą wspomniana już różnica w zawartości przestrzeni adresowej obu procesów: funkcja fork w procesie macierzystym zwraca wartość dodatnią (jest to numer utworzonego procesu potomnego), w procesie potomnym natomiast zwraca 0. Dodatkowo, niemożność wykonania rozwidlenia (na skutek zwykle osiągnięcia limitu liczby procesów) sygnalizowana jest wartością -1.

Przykład 3 (fork2.c) *Oto przerobiony kod poprzedniego przykładu, w którym proces macierzysty i potomny generują na standardowe wyjście różne znaki.*

```
#include <stdio.h>
#include <unistd.h>

void loop(char c)
```

```
{
    for (;;) {
        printf("%c", c);
        fflush(stdout);
        sleep(1);
    }
}

main()
{
    if (fork() > 0)
        loop('*');
    else
        loop('#');
}
```

Zadanie 3 *Dlaczego znaki * i # nie pojawiają się dokładnie naprzemiennie? Dlaczego dokładność naprzemienności wzrasta wraz z długością czasu pracy programu?*

Przestrzenie adresowe procesów macierzystego i potomnego są całkowicie rozłączne. Stwarza to pewne problemy w analizie wartości zmiennych.

Przykład 4 (fork3.c) *Proces macierzysty zajmuje się generowaniem z częstotliwością raz na sekundę na standardowe wyjście wartości zmiennej x. Proces potomny zwiększa tą zmienną z tą samą częstotliwością. Na standardowym wyjściu pojawiają się jednak same zera – proces macierzysty wyświetla wartość „swojego" x-a, który zmianom nie podlega.*

```
#include <stdio.h>
#include <unistd.h>

main()
{
    int x = 0;
    if (fork() > 0)
        for (;;) {
            printf("%d\n", x);
            sleep(1);
    } else
```

```
        for (;;) {
            x++;
            sleep(1);
        }

}
```

Pewne zasoby systemowe są jednak wspólne dla obu procesów. Są nimi między innymi deskryptory otwartych plików.

Przykład 5 (fork4.c) *Program czyta po jednym znaku z pliku tekstowego fork4.c, aż do jego końca. Plik został otworzony, a następnie program się rozwidlił, także kod odczytujący zawartość pliku jest wykonywany współbieżnie przez dwa procesy. Proces potomny różni się od procesu macierzystego niewypisywanie przeczytanego znaku na standardowe wyjście. Efekt? Proces potomny „kradnie" macierzystemu średnio co drugi znak.*

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main()
{
    char c;
    int fd = open("fork4.c", O_RDONLY), pid = fork();
    while (read(fd, &c, 1) == 1) {
        if (pid > 0) {
            printf("%c", c);
            fflush(stdout);
        }
        sleep(1);
    }
    return 0;
}
```

Zadanie 4 *Jak należałoby przerobić kod przykładu fork4.c, aby oba procesy czytały wszystkie znaki pliku?*

3.3 Koniec procesu potomnego

Zakończenie pracy procesu potomnego powoduje wysłanie do procesu macierzystego sygnału SIGCLD oraz przejście w stan zwany *zombie*. Wszystkie zasoby przydzielone procesowi potomnemu są zwalniane (np. zamykane są pliki), ale pozostaje wpis w liście procesów, mogący doprowadzić do jej przepełnienia. Wpis ten zostaje usunięty, o ile proces macierzysty wykona funkcję z rodziny *wait*. Najprostszą formą jest funkcja z argumentem typu wskazanie na int, gdzie zapisuje się stan zakończonego procesu potomnego. Kod powrotu procesu na przykład można uzyskać za pomocą makra WEXITSTATUS (definicja w pliku nagłówkowym sys/wait.h). Funkcja wait zwraca numer procesu potomnego.

Przykład 6 (wait1.c) *Proces potomny generuje 10 znaków * na standardowe wyjście przez 10 sekund a następnie kończy działanie z kodem wyjścia 7. Proces macierzysty nie robi nic poza zablokowaniem się w funkcji wait do czasu śmierci potomka. Wypisuje wtedy jego kod powrotu i sam również kończy działanie.*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

main()
{
    int status;
    if (fork() == 0) {
        int n;
        for (n = 0; n < 10; n++) {
            printf("*");
            fflush(stdout);
            sleep(1);
        }
        exit(7);
    }
    wait(&status);
    printf("\nKod powrotu %x\n", WEXITSTATUS(status));
    return 0;
}
```

Sprawdź zachowanie programu po zastąpieniu wywołania `wait(&status);` *przez* `sleep(10000);`. *Obserwuj listę procesów*

poprzez wydawanie polecenia `ps u`. *Kolumna STAT określa stan procesu, Z to zombie.*

Stosunkowo rzadko funkcja wait jest satysfakcjonującym rozwiązaniem obsłużenia zakończenia pracy procesu potomnego. Po pierwsze: jest ona blokująca co *de facto* oznacza, że proces macierzysty nie może robić nic poza oczekiwaniem na kończenie się procesów potomnych. Po drugie: jeżeli w jednej chwili zakończą się dwa procesy potomne, wait może pominąć jeden z nich. Po trzecie: nie możemy czekać na zakończenie konkretnego potomka. Aby rozwiązać te problemy, stosuje się zamiast wait funkcję *waitpid*. Jej pierwszy argument oznacza numer procesu potomnego, na którego zakończenie chcemy czekać (-1 oznacza tu dowolny proces). Drugi argument ma identyczne znaczenie jak w wait. Trzeci argument to opcje działania waitpid. Najczęściej używa się tu stałej WNOHANG, oznaczającej czekanie nieblokujące. Funkcja zwraca numer procesu lub -1 jeżeli żaden proces potomny się nie zakończył. Klasyczne rozwiązanie z waitpid w procedurze obsługi SIGCLD przedstawia przykład.

Przykład 7 (wait2.c) *Program macierzysty pracuje w pętli nieskończonej, w której użytkownik wprowadza liczby całkowite. Wprowadzenie liczby niedodatniej kończy program. Wprowadzenie liczby dodatniej n powoduje utworzenie procesu potomnego, który wyprowadza na standardowe wyjście co sekundę znak ASCII o kodzie 64+n (dla n=1 - litera A, dla n=2 - litera B itd.) n razy.*

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void handler_sigcld(int sig)
{
    int status, pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("[zakończono proces o pid=%d]", pid);
        fflush(stdout);
    }
}

void childloop(int n)
{
    int i;
```

```
    for (i = 0; i < n; i++) {
        printf("%c", 64 + n);
        fflush(stdout);
        sleep(1);
    }
    exit(0);
}

main()
{
    int n;
    signal(SIGCLD, handler_sigcld);
    for (;;) {
        scanf("%d", &n);
        if (n <= 0)
            break;
        if (fork() == 0)
            childloop(n);
    }
    return 0;
}
```

Zadanie 5 *W poprzednim przykładzie spróbuj zastąpić funkcję waitpid przez wait. Wprowadzając ciąg np.* `3 3 3 3 3 3 3[Enter]` *spowoduj utworzenie wielu procesów potomnych kończących się niemal równocześnie. Czym się objawiają losowe dysfunkcje programu?*

3.4 Funkcje tworzące nowy proces z kodu binarnego

Podstawową funkcją ładującą kod binarny aplikacji z pliku i uruchamiającą go jest *execve*. Przestrzeń adresowa procesu wykonującego funkcję execve jest zastępowana nowoutworzoną na podstawie danych z pliku wykonywalnego. A zatem kod aplikacji umieszczony za (udanym) wykonaniem execve nie ma już żadnego znaczenia.

Przykład 8 (exec1.c) *Program wykonuje kod zawarty w pliku wykonywalnym* ls *(aplikacja wyświetlająca zawartość katalogu, tu wywołana z argumentem* -l*). Na-*

pis „stop" nie pojawi się, chyba że execve się nie powiedzie (spróbuj np. zmienić nazwę aplikacji).

```
#include <stdio.h>
#include <unistd.h>

main(int argc, char *argv[], char *envp[])
{
    char *a[] = { "/bin/ls", "-l", NULL };
    printf("Start\n");
    execve(a[0], a, envp);
    printf("Stop\n");
    return 0;
}
```

Warto zwrócić uwagę na kilka subtelności w użyciu execve:

- execve nie wyszukuje pliku aplikacji w ścieżce dostępu (zmienna środowiska PATH);
- listy argumentów (drugi argument execve) i lista zmiennych środowiska (w formacie NAZWA=WARTOŚĆ, trzeci argument) są zakończone wskaźnikiem pustym;
- lista argumentów rozpoczyna się nie od pierwszego argumentu, a od nazwy pliku wykonywalnego (analogicznie jak w liście argumentów funkcji main).

Plik unistd.h dostarcza prototypów łatwiejszych w użyciu opakowań dla funkcji execve o nazwach *execl*, *execlp*, *execle*, *execv* oraz *execvp*. Funkcje execl... pozwalają zamiast używania tablicy łańcuchów argumentów specyfikować je jako swoje kolejne argumenty. Funkcja execle pozwala dodatkowo określić nowe środowisko. Funkcje exec...p dokonują wyszukania pliku aplikacji w ścieżce dostępu na podobieństwo shella.

Bardzo charakterystycznym motywem w programowaniu unixowym jest wywołanie *fork-exec/wait*. Proces rozwidla się, proces potomny uruchamia kod z pliku binarnego poprzez którąś z funkcji exec..., podczas gdy proces macierzysty oczekuje na jego zakończenie poprzez wywołanie wait.

Przykład 9 (exec2.c) *Aplikacja ta jest najprostszym interpreterem poleceń, pozwalającym na uruchomienie kodu innych aplikacji.*

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

main(int argc, char *argv[], char *envp[])
{
    char *a[64];                /* maksymalna liczba
                                   argumentów polecenia to 62 */
    char l[256];                /* maksymalna długość
                                   linii poleceń to 255 */
    int n;
    for (;;) {
        printf("co wykonac? ");
        fflush(stdout);
        if (fgets(l, 256, stdin) == NULL)
            break;
        a[0] = strtok(l, " \t\n\r");    /* pobranie pierwszego
                                           wyrazu linii */
        if (a[0] == NULL)
            continue;
        n = 1;
        while (a[n] = strtok(NULL, " \t\n\r"))
            n++;                /* pobranie kolejnych wyrazów */
        if (fork() == 0) {
            execvp(a[0], a);
            perror("execvp");   /* tylko w przypadku błędu */
            exit(0);
        } else
            wait(&n);
    }
    return 0;
}
```

Zadanie 6 *Jak zmieniło by się działanie poprzedniego programu, gdyby funkcja execvp wywołana by była bez uprzedniego fork?*

Wysokopoziomowych opakowań dla funkcji exec... dostarcza standardowa biblioteka języka C. W pliku nagłówkowym stdlib.h znajduje się prototyp funkcji *system*, realizującej model fork-exec/wait. Argumentem funkcji system jest łańcuch znaków - polecenie shellowe.

Zadanie 7 *Przetwórz kod poprzedniego przykładu, zastępując wywołania funkcji fork, execvp i wait przez system.*

Funkcja *popen* (stdio.h) z kolei wykonuje zadania funkcji system, dodatkowo wiążąc standardowe wejście lub wyjście uruchamianego programu z deskryptorem otwartego pliku w procesie macierzystym. Mówiąc inaczej: proces otwiera wirtualny plik - źródło danych, w którym pojawiają się one poprzez generowanie ich przez inną aplikację na standardowe wyjście lub też proces otwiera wirtualny plik, do którego zapisywane dane są konsumowane przez inną aplikację ze standardowego wejścia.

Przykład 10 (exec3.c) *Program odczytuje poprzez wywołanie polecenia* uname *nazwę systemu operacyjnego stacji roboczej.*

```
#include <stdio.h>

main()
{
    FILE *f;
    char b[256];
    f = popen("uname", "r");
    fgets(b, 256, f);
    printf("Twoj system to %s", b);
    pclose(f);
    return 0;
}
```

Zadanie 8 *Napisz program, który wywołany z pierwszym argumentem będącym nazwą pliku, wyrzuci go na standardowe wyjście po przetworzeniu przez filtr podany w drugim argumencie. Wskazówka: użyj wywołania* `popen(argv[2], "w")`.

3.5 Procesy - demony

Demonem (ang. *daemon* czyli duszek) nazywamy proces działający w tle, nie podlegający sterowaniu z żadnego terminala, uruchamiany zwykle podczas startu systemu i działający do jego zamknięcia. Demon świadczy zwykle usługi o charakterze systemowym. Ponieważ demony nie są związane z żadnym terminalem, wyjściem diagnostycznym dla nich jest zwykle podsystem dziennika pracy

systemu (demon *syslogd*), zwyczajowo zapisujący komunikaty w plikach tekstowych. Sterowanie demonami odbywa się poprzez sygnały lub nazwane środki komunikacji międzyprocesowej (patrz następny rozdział).

Zamiana zwykłego procesu na demona wiąże się z wykonaniem kilku czynności:

- Należy odłączyć proces od grupy terminala z którego został uruchomiony, czyniąc go *liderem* nowej grupy procesów (sesji). Zwykle wykorzystuje się do tego funkcję *setsid*.
- Należy zamknąć wszystkie pliki, włącznie ze standardowym wejściem, wyjściem i wyjściem dla błędów.
- Należy zmienić katalog roboczy tak, aby awaryjne zrzuty pamięci były tworzone w odpowiednim miejscu.
- Należy dokonać (często dwukrotnie) rozwidlenia wraz z zakończeniem procesu macierzystego. Wynikiem finalnym powinna być adopcja naszego procesu przez proces *init*.

Ponieważ czynności te w poszczególnych wersjach systemu Unix różnią się między sobą, twórcy bibliotek standardowych systemu często dostarczają funkcji (o nazwach *daemon* czy *daemonize*), które przeprowadzają wszystkie niezbędne czynności.

Przykład 11 (daemon1.c) *Oto szkieletowy program demona w systemie Linux.*

```
#include <unistd.h>
#include <signal.h>
#include <syslog.h>

void service_up(int quiet)
{
    if (!quiet)
        syslog(LOG_NOTICE, "Service started");
    sleep(1000000);
}

void service_down(int sig)
{
    if (sig)
```

```
        syslog(LOG_NOTICE, "Service stopped");
}

void service_reinit(int sig)
{
    service_down(0);
    syslog(LOG_NOTICE, "Service reinitialized");
    service_up(1);
}

main(int argc, char *argv[])
{
    daemon(0, 0);
    signal(SIGHUP, service_reinit);
    signal(SIGTERM, service_down);
    service_up(0);
    return 0;
}
```

Program ten w sposób charakterystyczny dla większości demonów systemowych obsługuje sygnały HUP (przeładowanie po np. zmianie konfiguracji) oraz TERM (zakończenie pracy). Komunikaty diagnostyczne demona można znaleźć (zależnie od konfiguracji demona syslogd) w pliku /var/log/user.log.

Sterowanie pracą demona jest możliwe za pomocą poleceń `killall -HUP nazwa` *i* `killall -TERM nazwa`.

Rozdział 4

Komunikacja międzyprocesowa

4.1 Łącza nienazwane

Funkcja *pipe* tworzy dwa deskryptory (identyfikatory otwartych plików). Pierwszy z nich służy do czytania, drugi do zapisywania danych. Ponieważ otwarte pliki są dziedziczone po rozwidleniu, dokonując asymetrycznego zamknięcia deskryptorów w procesie macierzystym i potomnym tworzymy jednokierunkowy kanał komunikacyjny, obsługiwany za pomocą standardowych funkcji plikowych.

Przestrzeń nazewnicza deskryptorów jest lokalna w ramach procesu i jego potomków, stąd użyteczność łącz nienazwanych jest ograniczona do procesów, których przodkiem jest proces tworzący łącze.

Przykład 12 (ipc1.c) *Proces macierzysty odpowiada za interakcje z użytkownikiem, proces potomny za obliczenia (symulowane przez funkcję sleep). Użytkownik wprowadza ze standardowego wejścia liczby całkowite. Liczby te są za pomocą łącza przesyłane do procesu potomnego, który wykonuje tyle dokładnie pętli wypisujących znali o kodzie ASCII 64+n, jedną na sekundę.*

```
#include <stdio.h>
#include <unistd.h>

main()
{
    int n, i;
    int p[2];
    pipe(p);
```

```
    if (fork() == 0) {
        close(p[1]);            /* proces potomny czyta */
        for (;;) {
            read(p[0], &n, sizeof(int));
            for (i = 0; i < n; i++) {
                printf("%c", 64 + n);
                fflush(stdout);
                sleep(1);
            }
        }
    } else {
        close(p[0]);            /* proces macierzysty pisze */
        for (;;) {
            printf("parent: ");
            fflush(stdout);
            scanf("%d", &n);
            write(p[1], &n, sizeof(int));
        }
    }
}
```

Program pozwala wprowadzać liczby równolegle do wykonywanego przetwarzania w procesie potomnym. Liczby te sa sukcesywnie zczytywane z łącza.

Zadanie 9 *Co dzieje się gdy po rozwidleniu nie zostaną zamknięte odpowiednie połówki łącz?*

Zadanie 10 *Rozbuduj (przez zestawienie drugiego łącza) program z ostatniego przykładu w ten sposób, aby proces macierzysty po wysłaniu liczby do procesu potomnego oczekiwał na dotarcie od niego informacji o zakończeniu działania w postaci tej samej liczby.*

Łącza mają ograniczoną pojemność (standard POSIX definiuje, że nie może być ona mniejsza niż 512B, zwykle używana jest pojemność 4kB). Jeżeli dane nie są z łącza odbierane w odpowiednim tempie, funkcja write nadawcy jest blokowana. Ograniczenie to dotyczy również łącz nazwanych z następnego rozdziału.

4.2 Łącza nazwane (FIFO)

Łącza nazwane pozwalają na usunięcie ograniczenia braku trwałej i dostępnej nazwy, pozwalającej na komunikację procesów niespokrewnionych. Przy okazji stwarzają możliwości ustalenia praw do korzystania z kanału komunikacyjnego.

Funkcja systemowa *mkfifo* czy też polecenie o tej samej nazwie wytwarza w systemie plików obiekt typu *named pipe*. Obsługa tego pliku specjalnego nie różni się od plików zwykłych, z wyjątkiem logiki odczytu i zapisu. Zapis do łącza nazwanego powoduje odłożenie informacji w buforze łącza (w jądrze systemu), odczyt - pobranie tej informacji.

Zadanie 11 *Wytwórz łącze nazwane* ipc1 *poleceniem* `mkfifo ipc1`. *Przerób kod przykładu z poprzedniego rozdziału, tak aby korzystał z łącza nazwanego.*

4.3 Podstawy IPC Systemu V

Opisane dalej formy komunikacji międzyprocesowej mają dwie wiodące implementacje: zgodną z normą POSIX oraz implementację z Systemu V. Ponieważ różnice w obu implementacjach są niewielkie, przedstawiono tu jedynie wersję bardziej rozpowszechnioną: interfejs Systemu V kolejek komunikatów, pamięci wspólnej i semaforów. Elementem nie należącym do tego interfejsu, niemniej wartym uwagi przez częste stosowanie przez programistów jest odwzorowanie pliku w pamięci, wymienione tu jako alternatywna metoda uzyskania pamięci wspólnej.

Wszystkie obiekty IPC Systemu V: kolejki komunikatów, bloki pamięci wspólnej oraz semafory używają jednolitej w systemie przestrzeni nazewniczej. W odróżnieniu jednak od rozwiązań posiksowych, nie jest to przestrzeń nazw systemu plików. Kluczem do obiektu IPC jest wartość typu całkowitego *key_t*, conajmniej 32-bitowego. Jako że posługiwanie się w nazewnictwie obiektów trwałych w systemie (a takimi są obiekty IPC Systemu V) liczbami całkowitymi jest niewygodne, dostarcza się programiście funkcji *ftok* (sys/ipc.h), wyliczającej na podstawie nazwy istniejącego pliku i dodatkowego 8-bitowego argumentu całkowitego (np. wersji programu) unikalną wartość typu key_t.

Przykład 13 (ftok1.c) *Program wypisuje wartość zwracaną przez funkcję ftok dla podanych w argumentach wywołania nazwy pliku i numeru wersji.*

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/ipc.h>

main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr,
                "uzyj: %s nazwapliku nrwersji\n", argv[0]);
        exit(1);
    }
    printf("%ld\n", ftok(argv[1], atoi(argv[2])));
    return 0;
}
```

Zwykle jako nazwę pliku dla funkcji ftok podajemy nazwę jakiego uwspólnionego między procesami pliku z danymi, wspólnego pliku konfiguracyjnego albo pliku z kodem binarnym aplikacji.

Obiekt IPC: kolejkę komunikatów, blok pamięci wspólnej czy zbiór semaforów tworzymy za pomocą funkcji *XXXget*, gdzie XXX jest odpowiednio równe *msg* dla kolejek komunikatów, *shm* dla pamięci dzielonej i *sem* dla semaforów. Jako pierwszy argument funkcja pobiera klucz, jako ostatni - sposób otwarcia (sumę bitową flag IPC_CREAT i IPC_EXCL z sys/ipc.h oraz praw dostępu, analogicznych do praw plikowych, niezbędnych przy tworzeniu nowego obiektu). Funkcja ta zwraca liczbę całkowitą będącą odpowiednikiem deskryptora otwartego pliku, wykorzystywaną w dalszych operacjach na obiekcie.

Jeżeli zamiast klucza podamy funkcji XXXget pierwszy argument równy IPC_PRIVATE, stworzymy obiekt IPC dostępny jedynie w zbiorze procesów potomnych naszego procesu (czyli tych, które znają deskryptor obiektu).

Funkcja *XXXctl* pozwala na wykonanie na odpowiednim obiekcie IPC operacji usunięcia z systemu lub zmienia jego właściwości. Pierszym argumentem jest tu deskryptor obiektu, drugim - rodzaj operacji (np. IPC_RMID dla usunięcia), trzecim - argument (dla usunięcia zwyczajowo 0).

Program użytkowy *ipcs* wyświetla listę obiektów IPC, program *ipcrm* natomiast usuwa obiekt o kluczu podanym w argumencie.

4.4 Kolejki komunikatów

Kolejkę komunikatów tworzymy za pomocą funkcji *msgget* (sys/msg.h), bez dodatkowych argumentów.

Komunikat jest dowolną strukturą o pierwszym polu typu long, o wartości większej od zera. Pole to zawiera typ komunikatu, ustalany przez programistę.

Wysłanie komunikatu do kolejki następuje za pomocą funkcji *msgsnd* z kolejnymi argumentami: deskryptor kolejki, adres komunikatu w pamięci, rozmiar komunikatu bez pierwszego pola oraz opcje przesłania (jeżeli są potrzebne).

Przykład 14 (msg1.c) *Program tworzy kolejkę komunikatów i zapisuje w niej trzy łańcuchy znaków o różnych długościach, po jednym w komunikacie typu 1.*

```
#include <string.h>
#include <sys/msg.h>

struct textmsg {
    long type;
    char text[16];
} msgs[] = { {
1, "I"}, {
1, "love"}, {
1, "you"}};

main()
{
    int d, n;
    d = msgget(ftok("msg1.c", 1), IPC_CREAT | 0666);
    for (n = 0; n < 3; n++)
        msgsnd(d, &(msgs[n]), strlen(msgs[n].text), 0);
    return 0;
}
```

Sprawdź po wykonaniu wynik polecenia `ipcs`.

Zadanie 12 *Co dzieje się po kolejnych wywołaniach programu?*

Funkcja *msgrcv* pobiera komunikat z kolejki. Podajemy jej w argumentach deskryptor kolejki, bufor na odczytany komunikat, rozmiar bufora (czyli wielkość maksymalnego możliwego do pobrania komunikatu) bez pierwszego pola, typ pobrania oraz flagi (jeśli są potrzebne).

Typ pobrania jest liczbą całkowitą interpretowaną następująco:

- dla 0 pobierany jest najstarszy komunikat (porządek FIFO);
- dla wartości dodatniej pobierany jest najstarszy komunikat tego właśnie typu;
- dla wartości ujemnej pobierany jest najstarszy komunikat o najmniejszym typie nie przekraczającym wartości bezwzględnej argumentu.

Funkcja zwraca liczbę bajtów w komunikacie, również bez uwzględnienia pierwszego pola.

Przykład 15 (msg2.c) *Program w pętli nieskończonej komunikaty wysyłane przez program z poprzedniego przykładu i wyświetla je na standardowym wyjściu.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>

struct textmsg {
    long type;
    char text[16];
};

main()
{
    struct textmsg msg;
    int d, n;
    if ((d = msgget(ftok("msg1.c", 1), 0)) < 0) {
        fprintf(stderr, "Brak kolejki\n"
                "Uruchom poprzedni program\n");
        exit(1);
    }
    for (;;) {
        n = msgrcv(d, &msg,
                   sizeof(struct textmsg) -
                   sizeof(long), 0, 0);
        msg.text[n] = 0;
        printf("Typ: %ld, tresc: %s\n", msg.type, msg.text);
```

```
    }
}
```

Zwróć uwagę na to, że funkcja msgrcv z flagą 0 ma charakter blokujący.

Zadanie 13 *Napisz program, który wysyła do wykorzystywanej poprzednio kolejki komunikatów łańcuchy wczytane ze standardowego wejścia. Zaraz po uruchomieniu, należy podać liczbę całkowitą, która będzie używana jako typ wszystkich dalszych komunikatów. Użyj programu z poprzedniego przykładu, aby sprawdzić działanie. Co przypomina ci w działaniu ten program?*

4.5 Pamięć wspólna

Funkcja *shmget* (sys/shm.h) tworzy nowy blok pamięci wspólnej. Jej drugim argumentem jest rozmiar bloku. Nowoutworzony blok jest wypełniany zerami.

Funkcja *shmat* dołącza blok pamięci wspólnej do przestrzeni adresowej procesu. Istnieje możliwość określenia adresu tego dołączenia ale zalecane jest pozostawić to systemowi, podając jako drugi argument shmat wskaźnik pusty. Argument trzeci to opcje dołączenia (zwykle 0).

Przykład 16 (shm1.c) *Program tworzy blok pamięci wspólnej zdolny przechować 10-elementową tablicę wartości typu int, a następnie tworzy dwa procesy potomne, które co sekundę podnoszą wartości w odpowiednio parzystych i nieparzystych komórkach tablicy. Proces główny zajmuje się wypisywaniem wartości z tablicy na standardowe wyjście.*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>

#define TSIZE 10

void inc(int T[], int off)
{
    int n;
    for (;;) {
        for (n = off; n < TSIZE; n += 2)
            T[n]++;
```

```
        sleep(1);
    }
}

main(int argc, char *argv[])
{
    int d, n;
    int *T;
    d = shmget(ftok(argv[0], 1),
               TSIZE * sizeof(int), IPC_CREAT | 0600);
    T = shmat(d, NULL, 0);
    if (fork() == 0)
        inc(T, 0);
    if (fork() == 0)
        inc(T, 1);
    for (;;) {
        for (n = 0; n < TSIZE; n++)
            printf("%3d ", T[n]);
        printf("\n");
        sleep(1);
    }
}
```

Ponieważ wszystkie trzy procesy mniej więcej w tym samym czasie rozpoczynają cykliczne wykonywanie swoich zadań, możemy zauważyć że czasami proces zwiększający pozycje parzyste wyprzedza zwiększającego pozycje nieparzyste, a czasami odwrotnie. Pamięć wspólna jest trwała w systemie, więc po zatrzymaniu programu przez Ctrl+C i powtórnym uruchomieniu wzrosty rozpoczynają się od ostatnich wartości.

Zwróć uwagę, że mimo tego, że procesy potomne mają rozłączne przestrzenie adresowe, nie jest wymagane, aby wywoływały one funkcję shmat.

4.6 Odwzorowanie pliku w pamięci

Fundamentem tej metody jest funkcja *mmap* (sys/mman.h). Funkcja ta tworzy odwzorowanie w pamięci otwartego pliku. Wszystkie zmiany w odwzorowanym fragmencie pamięci są synchronizowane z zawartością pliku.

Funkcji mmap podajemy kolejno jako argumenty: wartość wskaźnika na który

ma być odwzorowany plik (zwyczajowo podajemy tu NULL, aby system sam dobrał wartość wskaźnika), rozmiar odwzorowywanej pamięci (zwykle mniejszą lub równą rozmiarowi pliku), sposób dostępu do pamięci (jako sumę bitową flag PROT_READ i PROT_WRITE, kompatybilną z trybem otwarcia pliku), sposób korzystania z pamięci (zwykle używamy tu stałej MAP_SHARED), otwarty deskryptor pliku oraz przesunięcie odwzorowania względem jego początku.

Przykład 17 (map1.c) *Program tworzy plik o nazwie* `/tmp/shm1` *i zapisuje do niego 1000 bajtów o wartości 0. Następnie odwzorowuje ten plik w pamięci i w pętli nieskończonej zwiększa co sekundę kolejne bajty odwzorowanej pamięci.*

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define SHMFILENAME "/tmp/shm1"
#define SHMSIZE 1000

main()
{
    int shmd, n;
    char *shm;
    char zero = 0;

    shmd = open(SHMFILENAME, O_CREAT | O_RDWR, 0666);
    for (n = 0; n < SHMSIZE; n++)
        write(shmd, &zero, 1);
    shm =
        (char *) mmap(NULL, SHMSIZE,
                      PROT_READ | PROT_WRITE,
                      MAP_SHARED, shmd, 0);

    n = 0;
    for (;;) {
        shm[n]++;
        if (++n >= SHMSIZE)
            n = 0;
        sleep(1);
    }
}
```

Rezultat działania programu można śledzić na bieżąco poleceniem `od -b /tmp/shm1`

Zadanie 14 *Odwzoruj w pamięci plik z tekstem źródłowym programu z przykładu i odczytaj jego treść za pomocą pętli wypisującej elementy stosownej tablicy.*

Istotą wykorzystania funkcji mmap w kreowaniu międzyprocesowej pamięci wspólnej jest odwzorowanie w pamięci dwóch procesów tego samego pliku.

Przykład 18 (map2.c) *Program odwzorowuje plik* `/tmp/shm1` *używany w poprzednim przykładzie na własną 1000-elementową tablicę bajtów, dostępną wyłącznie do odczytu. Co sekundę wyświetla jej zawartość (dokładniej ostatnią cyfrę każdego bajtu) w formacie 50x20.*

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define SHMFILENAME "/tmp/shm1"
#define SHMSIZE 1000

main()
{
    int shmd, n, i, j;
    char *shm;

    shmd = open(SHMFILENAME, O_RDONLY);
    shm = (char *) mmap(NULL, SHMSIZE, PROT_READ,
                        MAP_SHARED, shmd, 0);

    for (;;) {
        for (i = 0; i < 25; i++)
            printf("\n");
        n = 0;
        for (i = 0; i < 20; i++) {
            for (j = 0; j < 50; j++) {
                printf("%d", shm[n] % 10);
                n++;
```

```
            }
            printf("\n");
        }
        sleep(1);
    }
}
```

Ponieważ system plików jest udostępniany wszystkim procesom, wspólna pamięć może być odwzorowywana w procesach niespokrewnionych, również mających różnych właścicieli. Jeżeli nie zależy nam na tym i chcemy jedynie mieć wspólny bufor pamięci między procesem macierzystym a potomnym, możemy odwzorować za pomocą funkcji mmap blok anonimowy (styl systemu BSD) lub urządzenie /dev/zero (styl Systemu V).

Przykład 19 (map3.c) *Program tworzy anonimowy blok wspólnej pamięci zdolny przetrzymać jedną wartość typu long i rozwidla się. Proces potomny ustawicznie zwiększa współdzieloną wartość, którą proces macierzysty odczytuje co sekundę.*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>

main()
{
    unsigned long *shm;

    shm = (unsigned long *) mmap(NULL, sizeof(long),
                                 PROT_READ |
                                 PROT_WRITE,
                                 MAP_SHARED | MAP_ANON,
                                 ~0, 0);
    *shm = 0;
    if (fork() == 0) {
        for (;;)
            (*shm)++;
    }
    for (;;) {
        printf("\r%010lu", *shm);
        fflush(stdout);
```

```
        sleep(1);
    }
}
```

Zwróć uwagę na przedostatni argument - zamiast deskryptora otwartego pliku pojawia się tu bitowe zaprzeczenie zera, wartość nieosiągalna dla deskryptorów plikowych.

Przewagą odwzorowań plików w pamięci nad pamięcią wspólną Systemu V jest możliwość zmiany rozmiaru bloku wspólnego poprzez zmianę rozmiaru odwzorowywanego pliku dzięki funkcji *ftruncate*.

4.7 Semafory

Współbieżne operacje na pamięci grożą jej desynchronizacją. Zjawisko to powstaje wtedy, gdy pewna podzielna (wywłaszczalna) operacja modyfikująca jest wykonywana przez wiele procesów jednocześnie. Po wywłaszczeniu jednego procesu w jej trakcie, drugi dostaje dane częściowo tylko zmienione - zwykle nieprawidłowe.

Przykład 20 (sem1.c) *Program pracuje na 10-elementowej tablicy liczb całkowitych, wypełnionej wartościami 0, 1, 2 do 9. Proces główny tworzy kilka procesów potomnych, które wykonują po 10000 rotacji tablicy, polegającej na przesunięciu wartości z pozycji n-tej na (n-1)-szą. Element zerowy staje się nowym elementem ostatnim. Po zakończeniu procesów potomnych, proces główny wyświetla zawartość tablicy.*

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/mman.h>

/* funkcja rotujaca tablice */
void rotate(int t[], int N)
{
    int n, tmp;
    tmp = t[0];
```

```
    for (n = 1; n < N; n++)
        t[n - 1] = t[n];
    t[N - 1] = tmp;
}

#define NCHILDREN 5
#define NROTATES 10000
#define TABSIZE 10

int nchildren = 0;

/* procedura obslugi sygnalu SIGCLD */
void handler_sigcld(int sig)
{
    int status;
    while (waitpid(-1, &status, WNOHANG) > 0)
        nchildren--;
}

main()
{
    int *shm;
    int n, i;
    int pids[NCHILDREN];

    /* instalacja procedury obslugi sygnalu SIGCLD */
    signal(SIGCLD, handler_sigcld);

    /* inicjalizacja tablicy w pamieci wspolnej */
    shm = (int *) mmap(NULL, TABSIZE * sizeof(int),
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_ANON, ~0, 0);
    for (n = 0; n < TABSIZE; n++)
        shm[n] = n;

    /* utworzenie NCHILDREN procesow potomnych */
    for (n = 0; n < NCHILDREN; n++) {
        if ((pids[n] = fork()) == 0) {
            for (i = 0; i < NROTATES; i++)
                rotate(shm, TABSIZE);
            exit(0);
```

```
        } else
            nchildren++;
    }

    /* oczekiwanie na zakonczenie procesow potomnych */
    while (nchildren > 0);

    /* wydrukowanie stanu tablicy */
    for (n = 0; n < TABSIZE; n++)
        printf("%d ", shm[n]);
    printf("\n");
    return 0;
}
```

Póki jest tylko jeden proces rotujący (NCHILDREN = 1), działanie programu jest intuicyjnie poprawne. Przy zwiększeniu liczby procesów potomnych, tablica traci cechę sekwencyjności wartości. Funkcja rotate *nie może być podzielna czyli tylko jeden z procesów potomnych może rotować tablicę w tym samym czasie.*

Zadanie 15 *Opracuj inny przykład, demonstrujący desynchronizację pamięci wspólnej.*

Część kodu, która musi być wykonywana niepodzielnie (atomowo) przez co najwyżej jeden z procesów grupy nazywa się zwykle *sekcją krytyczną*. Najpopularniejszą metodą zapewniania atomowości kodu jest zabezpieczenie go *semaforem*. Na semaforze możemy wykonać dwie operacje - możemy czekać pod zamkniętym semaforem (tradycyjnie nazywa się tą operację *P* lub *wait*) lub go podnieść (*V* lub *post*).

Semafory implementuje się zwykle jako współdzielone zmienne całkowite, których wartość niedodatnia oznacza stan zamknięty. P jest więc operacją oczekiwania na dodatnią wartość semafora, V operacją jego zwiększenia.

Jeżeli początkowa wartość semafora jest równa 1, mamy do czynienia z *semaforem binarnym*, jeżeli jest większa niż 1 - z *semaforem zliczającym*.

Operacjami P i V na semaforze binarnym otacza się sekcję krytyczną.

W IPC Systemu V możemy stworzyć od razu zbiór semaforów - o jego liczności decyduje drugi argument funkcji *semget* (sys/sem.h). Jeżeli używamy funkcji semget do używania istniejącego już zbioru semaforów, drugi argument nie ma znaczenia.

Nowoutworzone tablice semaforów są niezainicjowane - nie możemy zakładać, że semafory są zamknięte czy otwarte. Do ich inicjowania należy użyć funkcji *semctl*, która ma trzy obowiązkowe argumenty: deskryptor zbioru, indeks semafora w zbiorze (licząc od 0) oraz stałą operacji do wykonania. Operacjami tymi są:

- GETVAL zwraca wartość semafora;
- SETVAL nadaje wartość semafora;
- GETALL przepisuje wartości całej tablicy semaforów do zaalokowanej przez programistę tablicy elementów typu unsigned short;
- SETALL nadaje wartości całej tablicy semaforów według tablicy programisty.

Pozostałe, opcjonalne argumenty semctl są uniami:

```
union semun {
  int val;              /* istotne dla SETVAL */
  struct semd_ds *buf;
  ushort *array;        /* istotne dla GETALL i SETALL */
};
```

Ponieważ w większości współczesnych systemów rozmiary komponentów powyższej unii są takie same, wielu programistów używa jako dalszych argumentów semctl zamiast unii odpowiednich wartości i tak:

- `semctl(d, n, GETVAL)` zwraca wartość n-tego semafora w tablicy;
- `semctl(d, n, SETVAL, x)` ustawia wartość n-tego semafora w tablicy na x;
- `semctl(d, 0, GETALL, buf)` przepisuje zawartość tablicy semaforów do tablicy użytkownika buf;
- `semctl(d, 0, SETALL, buf)` przepisuje zawartość tablicy użytkownika buf do tablicy semaforów.

Operacje P i V są realizowane przez funkcję *semop*. Pierwszym jej argumentem jest deskryptor zbioru semaforów, drugim - zbiór operacji do niepodzielnego

wykonania na semaforach w postaci tablicy struktur `sembuf`, a trzecim liczność tej tablicy. Struktura sembuf zawiera przynajmniej trzy pola typu short: *sem_num* - numer semafora, *sem_op* - kod operacji na semaforze oraz *sem_flg* - opcje operacji.

W najprostszym przypadku programista powinien zadeklarować zmienną typu `struct sembuf`, o nazwie powiedzmy `op`, nadać wartości komponentom sem_num, sem_op i sem_flg i wywołać `semop(d, &op, 1)`.

Kluczowe znaczenie w pracy na semaforach ma wartość komponentu sem_op:

- Jeżeli sem_op jest dodatnia, to semafor jest zwiększany o tę wartość. Wartość 1 oznacza więc typową operację podniesienia semafora (V).
- Zerowa wartość sem_op uruchamia oczekiwanie na zerową wartość semafora. Nie odpowiada to żadnej typowej operacji na semaforze.
- Ujemna wartość sem_op uruchamia oczekiwanie na wartość semafora większą lub równą wartości bezwzględnej sem_op, po czym wartość sem_op jest dodawana do wartości semafora (jest on zmniejszany). Wartość -1 oznacza więc typową operację czekania pod semaforem (P).

Przykład 21 (sem2.c) *Program dokonuje rozwidlenia. Proces potomny generuje z częstotliwością raz na sekundę znak B na standardowe wyjście, w porcjach po 10 znaków. Proces macierzysty dokonuje analogicznych operacji ze znakiem A. Użycie semaforów powoduje, że porcje liter układają się ciągami po 10.*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>

struct sembuf P, V;

void gen(int d, char c)
{
    int n;
    semop(d, &P, 1);
    for (n = 0; n < 10; n++) {
        printf("%c", c);
        fflush(stdout);
        if (n < 9)
            sleep(1);
```

```
    }
    semop(d, &V, 1);
    sleep(1);
}

main(int argc, char *argv[])
{
    int d;
    d = semget(ftok(argv[0], 1), 1, IPC_CREAT | 0600);
    semctl(d, 0, SETVAL, 1);
    P.sem_num = 0;
    P.sem_op = -1;
    P.sem_flg = 0;
    V.sem_num = 0;
    V.sem_op = +1;
    V.sem_flg = 0;
    if (fork() == 0)
        for (;;)
            gen(d, 'B');
    for (;;)
        gen(d, 'A');
}
```

Mimo, że wygodniej byłoby tu zainicjować wartości zmiennych P i V statycznie, nie jest to zalecane, ponieważ standard nie definiuje kolejności występowania komponentów w strukturze sembuf.

Zadanie 16 *W powyższym przykładzie zakomentuj wywołania semop i obserwuj wyniki działania. Wyjaśnij rezultat.*

Zadanie 17 *W powyższym przykładzie zakomentuj ostatnie wywołanie funkcji sleep w funkcji gen. Wyjaśnij rezultat działania tak zmienionego programu.*

Zadanie 18 *Zsynchronizuj semaforami program z początku rozdziału.*

Rozdział 5

Wątki

5.1 Podstawy

Tradycyjny sposób tworzenia unixowego elementu aktywnego - rozwidlanie aktywnego procesu - ma kilka wad. Jedną z nich jest zasobożerność funkcji fork, nawet przy zastosowaniu mechanizmów tzw. *leniwego kopiowania* tj. wtedy gdy proces potomny zmienia swoją kopię przestrzeni adresowej. Drugą jest skomplikowany transfer danych między procesami rodzicielskim i potomnym. Łącza czy IPC Systemu V są dość skomplikowane w użyciu i słabo przystają do języków programowania wysokiego poziomu.

Standard POSIX definiuje, a większość współczesnych systemów unixowych implementuje interfejs zwany *posix threads* (*pthreads*), pozwalający na utworzenie w ramach tej samej przestrzeni adresowej, wielu elementów aktywnych - wątków. Z punktu widzenia tego interfejsu, tradycyjny ”ciężki” proces unixowy jest procesem z jednym wątkiem sterowania. Interfejs pthreads pozwala na utworzenie innych, w tej samej przestrzeni adresowej.

Programując w języku C z użyciem interfejsu pthreads, otrzymujemy czytelne odwzorowanie przestrzeni adresowych wątków na elementy programu. Wszystkie wątki procesu mają wspólny kod. Wspólne są również zmienne nieautomatyczne - globalne i statyczne oraz deskryptory, dane związane z sygnałami, otoczenie systemowe (np. katalog roboczy). Wyłączną własnością wątku jest natomiast stos czyli m.in. zmienne automatyczne oraz licznik rozkazów.

Wątki stwarzają pewne problemy z funkcjami standardowej biblioteki języka C, które zapisują dane w statycznych obszarach pamięci (np. strtok). Zaleca się używanie specjalnych wersji tych funkcji, przystosowanych do pracy w środowisku z wątkami.

Interfejs pthreads jest często umieszczany poza biblioteką standardową języka C, także kompilacja programów z użyciem wątków wymaga specjalnych dyrektyw dla programu łączącego (w przypadku linuxa i kompilatora gcc należy używać opcji `-lpthread`).

5.2 Tworzenie i kończenie wątku

Dla nowego wątku programista musi zdefiniować akcję, którą będzie wykonywał. Definicja ta jest funkcją pobierająca wskaźnik w argumencie (typu `void *`) i taki wskaźnik zwracająca. Za pomocą argumentu możemy nowotworzonemu wątkowi przekazać dane z wątku tworzącego, za pomocą wartości zwracanej możemy po jego zakończeniu jakieś dane uzyskać. Często jednakże możliwości te nie są wykorzystywane, jako że i tak nowy wątek współdzieli z wątkiem tworzącym zmienne globalne.

Funkcja tworząca wątek, *pthread_create* (pthread.h), pobiera w argumentach: wskaźnik do zmiennej typu *pthread_t* gdzie będzie zapisany identyfikator wątku (w odróżnieniu od identyfikatora procesu nie musi on mieć charakteru globalnego w systemie), wskaźnik do struktury *pthread_attr_t* zawierającej atrybuty nowego wątku (przy wartości NULL wątek tworzony jest z atrybutami domyślnymi), nazwę funkcji - akcji wątku oraz wskaźnik do argumentów.

Przykład 22 (thr1.c) *Program tworzy trzy nowe wątki, wykonujące tę samą procedurę, wypisującą w pętli nieskończonej co sekundę znak na standardowe wyjście. Po utworzeniu wątków, wątek główny również rozpoczyna generowanie znaku. Znak wypisywany przez akcję wątku jest przekazywany w argumencie.*

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *gen(void *arg)
{
    for (;;) {
        printf("%c", *((char *) arg));
        fflush(stdout);
        sleep(1);
    }
}
```

```
main()
{
    pthread_t t1, t2, t3;
    char s[] = ".ABC";
    pthread_create(&t1, NULL, gen, s + 1);
    pthread_create(&t2, NULL, gen, s + 2);
    pthread_create(&t3, NULL, gen, s + 3);
    gen(s + 0);
}
```

Pamiętaj o kompilacji programu z opcją `-lpthread`*!*

Zadanie 19 *Dlaczego w poprzednim przykładzie znaki pojawiają się w różnym porządku?*

Wątek kończy się gdy:

- kończy się funkcja wątku;
- wątek wywoła funkcję *pthread_exit* z argumentem w postaci wskaźnika - wartości zwrotnej;
- kończy się proces (wątek główny).

Wątek może być *przyłączalny* (ang. *joinable*) lub odłączony (ang. *detached*). Domyślny stan przyłączalny można zmienić wywołując funkcję *pthread_detach* z identyfikatorem wątku. Jeżeli stan odłączony chce osiągnąć wątek bieżący, jako argumentu należy użyć wartości zwracanej przez bezargumentową funkcję *pthread_self*.

Na zakończenie wątku przyłączalnego można oczekiwać za pomocą funkcji *pthread_join*, wywołanej z dwoma argumentami: identyfikatorem wątku oraz wskaźnikiem na miejsce w pamięci, gdzie ma zostać zapisana wartość zwracana funkcji wątku. Niestety, nie można oczekiwać na zakończenie wielu wątków, a tylko jednego konkretnego.

Przykład 23 (thr2.c) *Program tworzy trzy wątki, które kończą działanie po czasie odpowiednio 3, 5 i 8 sekund. Wątek główny oczekuje na ich zakończenie, mierząc czas.*

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

void *action(void *arg)
{
    sleep(*((int *) arg));
    return NULL;
}

main()
{
    pthread_t t1, t2, t3;
    int start;
    int tt[] = { 3, 5, 8 };
    void *res;
    pthread_create(&t1, NULL, action, tt);
    pthread_create(&t2, NULL, action, tt + 1);
    pthread_create(&t3, NULL, action, tt + 2);
    start = time(NULL);
    pthread_join(t1, &res);
    printf("%d\n", time(NULL) - start);
    pthread_join(t2, &res);
    printf("%d\n", time(NULL) - start);
    pthread_join(t3, &res);
    printf("%d\n", time(NULL) - start);
    return 0;
}
```

Zadanie 20 *W programie z poprzedniego przykładu odwróć kolejność czasów zakończeń wątków (na 8, 5, 3). Wyjaśnij uzyskany rezultat.*

Na zakończenie wątku w stanie odłączonym oczekiwać nie można. Działa on na podobieństwo demona.

5.3 Muteksy

Muteks (od angielskiego *mutual exclusion* czyli wzajemne wyłączanie; w polskiej literaturze spotyka się również określenie *zamek* lub *rygiel*) jest odpowiednikiem

semafora binarnego, działającym we wspólnej przestrzeni adresowej wątków procesu.

Muteksy w interfejsie pthreads są typu *pthread_mutex_t*. Deklaruje się je zwykle jako zmienne globalne, przed wykorzystującymi je funkcjami wątków. Muteks powinien, najlepiej w ramach deklaracji, być zainicjowany wartością *PTHREAD_MUTEX_INITIALIZER*. Muteks ma dwa stany: otwarty i zamknięty. Początkowo muteks jest w stanie otwartym (odpowiednik podniesionego semafora).

Muteks zamykamy funkcją *pthread_mutex_lock*, otwieramy *pthread_mutex_unlock* (obie funkcje za argument biorą wskaźnik na muteks). Co najwyżej jeden z wątków procesu może zamknąć muteks - pozostałe czekają na jego otwarcie. Muteks jest więc najprostszym środkiem zapewnienia wzajemnego wykluczania w sekcji krytycznej.

Przykład 24 (mut1.c) *Główny wątek programu dokonuje w pętli nieskończonej zmian wartości zmiennej globalnej x według algorytmu: wylosuj liczbę całkowitą r z zakresu 5–9, zapisz ją w zmiennej x, wstrzymaj pracę na sekundę, wyzeruj x. Zmienna x ma więc wartość dodatnią, większą lub równą 5 przez sekundę a następnie przez kilka sekund wartość 0 itd. Program tworzy dwa wątki, działające w oparciu o tę samą funkcję - w pętli nieskończonej następuje: oczekiwanie (aktywne) na niezerową wartość x, wstrzymanie pracy na odczytaną liczbę sekund oraz wygenerowanie tylu znaków A lub B (w zależności od tego, przez który wątek) ile sekund wątek był wstrzymany.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

int x = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *gen(void *arg)
{
    int n;
    for (;;) {
        while (x == 0);
        n = x;
        sleep(n);
```

```
        pthread_mutex_lock(&mutex);
        while (n-- > 0) {
            printf("%c", *((char *) arg));
            fflush(stdout);
            /* usleep(100); */
        }
        printf("\n");
        pthread_mutex_unlock(&mutex);
    }
}

main()
{
    pthread_t t1, t2;
    char s[] = "AB";
    int r;
    srand(time(NULL));
    pthread_create(&t1, NULL, gen, s);
    pthread_create(&t2, NULL, gen, s + 1);
    for (;;) {
        r = 5 + rand() % 5;
        printf("--- %d ---\n", r);
        x = r;
        sleep(1);
        x = 0;
        sleep(r);
    }
}
```

Zauważ, że program jest tak skonstruowany, że wątki generujące znaki na standardowe wyjście powinny zacząć to robić w tym samym czasie (teoretycznie). Muteks zabezpiecza przed zakłóceniem wyświetlania znaków przez jeden wątek drugiemu. Dokładniej: w programie przyjęto zasadę, że co najwyżej jeden wątek naraz wypisuje sekwencję liter. Jeżeli wywołania funkcji `pthread_mutex_lock` *i* `pthread_mutex_unlock` *zostałyby zakomentowane, nic nie chroni pętli wypisujących znaki przed przeplotem - co powien czas powinno się zaobserwować zdarzenie polegające na wymieszaniu liter A i B w linii (jeżeli nie możesz się tego doczekać, odkomentuj wywołanie funkcji* `usleep`*).*

Zadanie 21 *Zmień program tak, aby litery A pojawiały się zawsze przed literami B.*

Przykład 25 (mut2.c) *Wątek główny tworzy plik* `mut2.dat` *i zapisuje do niego (tekstowo) liczbę 0. Dwa nowoutworzone wątki dokonują po 1000 razy operacji: otwórz plik do odczytu, odczytaj liczbę, zamknij plik, zwiększ liczbę o 1, otwórz plik do zapisu, zapisz liczbę, zamknij plik. Program nie używa muteksów.*

```
#include <stdio.h>
#include <pthread.h>

#define FNAME "mut2.dat"

void *inc(void *arg)
{
    FILE *f;
    int x, n;
    for (n = 0; n < 1000; n++) {
        f = fopen(FNAME, "r");
        if (fscanf(f, "%d", &x)) {
            fclose(f);
            f = fopen(FNAME, "w");
            fprintf(f, "%d\n", x + 1);
        }
        fclose(f);
    }
}

main()
{
    pthread_t t1, t2;
    void *res;
    FILE *f;
    f = fopen(FNAME, "w");
    fprintf(f, "%d\n", 0);
    fclose(f);
    pthread_create(&t1, NULL, inc, NULL);
    pthread_create(&t2, NULL, inc, NULL);
    pthread_join(t1, &res);
    pthread_join(t2, &res);
    return 0;
}
```

Sprawdź zawartość pliku `mut2.dat` *po wykonaniu programu. To że plik*

zawiera liczbę 2000 jest dość mało prawdopodobne.

Zadanie 22 *Zabezpiecz program z poprzedniego przykładu przed błędami wynikającymi z braku zsynchronizowania wątków.*

Zadanie 23 *Przepisz kod przykładu 21 z użyciem wątków i muteksów.*

5.4 Zmienne warunkowe

Zmienne warunkowe wykonują zadania znanych z teorii systemów operacyjnych *monitorów*. W odróżnieniu od muteksów pozwalają na konstrukcje typu „oczekuj na zmianę warunku”.

Przykład 26 (cond1.c) *Wątek główny powoli (o 1 na sekundę) zwiększa wartość zmiennej globalnej. Kiedy wartość ta osiągnie 20, uaktywnia się drugi wątek, który również wykonuje to zwiększanie.*

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int x = 0;

void inc(int verbose)
{
    for (;;) {
        x++;
        if (verbose) {
            printf("\r%d", x);
            fflush(stdout);
        }
        sleep(1);
    }
}

void *waitfor20(void *arg)
{
    while (x < 20);
```

```
    inc(0);
}

main()
{
    pthread_t t;
    pthread_create(&t, NULL, waitfor20, NULL);
    inc(1);
}
```

Zaobserwuj (np. programem `top`*), że podczas pierwszych 20 sekund pracy programu, proces zawłaszcza całą dostępną moc obliczeniową procesora. Przyczyną tego jest* aktywne czekanie *w pierwszej linii funkcji* `waitfor20`*. Wątek nieustannie próbkuje stan zmiennej, aby ruszyć akcję niezwłocznie po spełnieniu żądanego warunku. Niepożądany efekt wirowania obliczeniowego można usunąć poprzez zastosowanie zmiennych warunkowych.*

Zmienna warunkowa jest obiektem typu *pthread_cond_t*. Podobnie jak muteks, zmienna warunkowa powinna być inicjowana za pomocą stałej *PTHREAD_COND_INITIALIZER*. Z każdą zmienną warunkową powinien być związany muteks, który zabezpiecza atomowość sprawdzenia warunku i wyjścia z pętli oczekiwania.

Na zmiennych warunkowych możemy dokonywać dwóch operacji: oczekiwania oraz sygnalizacji. Oczekiwanie jest stanem zablokowania (nieaktywnego) wątku do czasu spełnienia warunku. Sygnalizacja jest czynnością budzenia jednego lub więcej wątków.

Oczekiwanie realizujemy poprzez funkcję *pthread_cond_wait*, wywoływaną z dwoma wskaźnikami: na zmienną warunkową oraz na związany z nią muteks. Realizacja pthread_cond_wait kończy się na dwa sposoby: albo wątek zostaje wstrzymany i muteks zwolniony, albo wątek kontynuuje działanie przy zamkniętym muteksie. Sygnalizację zapewnia funkcja *pthread_cond_signal* (budzenie jednego wątku) lub *pthread_cond_broadcast* (budzenie wszystkich wątków zablokowanych w oczekiwaniu na spełnienie warunku). Obie wywołujemy ze wskaźnikami na zmienną warunkową. Typowy fragment kodu oczekiwania na spełnienie warunku wygląda następująco:

```
pthread_mutex_lock(&muteks);
while(!warunek)
  pthread_cond_wait(&zm_warunkowa, &muteks);
```

```
/* chroniona operacja, mogaca zmienic warunek */
pthread_mutex_unlock(&muteks);
```

Zadanie 24 *Zastosuj zmienną warunkową do programu z poprzedniego przykładu.*

Przykład 27 (cond2.c) *Wątek główny wczytuje ze standardowego wyjścia liczby całkowite (dopuszczalne są wartości z zakresu 1–10). Wprowadzenie liczby n z zakresu 1–5 powoduje obudzenie wątku generującego n liter A, wprowadzenie liczby n z zakresu 6–10 obudzenie wątku generującego n liter B.*

```
#include <stdio.h>
#include <pthread.h>

struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
} x = {
PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};

struct thrarg {
    char c;
    int min, max;
};

void *action(void *arg)
{
    int n, v;
    struct thrarg *a = (struct thrarg *) arg;
    for (;;) {
        pthread_mutex_lock(&x.mutex);
        while (!(x.value >= a->min && x.value <= a->max))
            pthread_cond_wait(&x.cond, &x.mutex);
        v = x.value;
        x.value = 0;
        pthread_mutex_unlock(&x.mutex);
        for (n = 0; n < v; n++)
            printf("%c", a->c);
        printf("\n");
```

```
    }
}

main()
{
    static struct thrarg a1 = { 'A', 1, 5 }, a2 = {
    'B', 6, 10};
    pthread_t t1, t2;
    pthread_create(&t1, NULL, action, &a1);
    pthread_create(&t2, NULL, action, &a2);
    for (;;) {
        scanf("%d", &x.value);
        pthread_cond_broadcast(&x.cond);
    }
}
```

Dość charakterystyczne jest użyte tu zgrupowanie w jednej strukturze muteksu, zmiennej warunkowej i aktorów warunku (tu wartości komponentu `value`*), które poprawia czytelność kodu.*

Rozdział 6

Gniazdka

6.1 Podstawy

Gniazdka (ang. *sockets*) są najbardziej podstawową formą komunikacji rozproszonej w systemie Unix. Interfejs ten jest tak skonstruowany, że komunikacja lokalna i międzymaszynowa jest realizowana za pomocą tych samych funkcji. Więcej: realizacja transferu danych (w trybie strumieniowym) odbywa się za pomocą standardowych funkcji plikowych read i write (tj. jak w łączach).

Gniazdka są programistycznym odwzorowaniem podstawowych protokołów transportowych TCP/IP. Interfejs jest tak skonstruowany, że istnieje możliwość jego adaptacji w rodzinie innych protokołów np. IPv6. Poniżej omówione jednak zostaną tylko dwie dziedziny: gniazdka lokalne (substytuty łącz i łącz nazwanych) oraz gniazdka w dziedzinie internetowej (IPv4).

6.2 Struktury adresowe

Plik nagłówkowy sys/socket.h zawiera deklarację ogólnej struktury adresowej *sockaddr*. Typu tego nie wykorzystujemy wprost. W każdej dziedzinie punkt komunikacji identyfikujemy inaczej, tak więc używamy różnych typów w dziedzinie lokalnej (*sockaddr_un* z sys/un.h) i internetowej (*sockaddr_in* z netinet/in.h). Kiedy jednak przekazujemy wskaźnik do struktury adresowej funkcji interfejsu gniazdek, musimy zrzutować go na wskaźnik do ogólnej struktury adresowej.

Struktury adresowe, zależnie od implementacji, mogą mieć różną budowę. Standard POSIX definiuje jedynie pola, które muszą wystąpić. Należy bez-

względnie pamiętać, aby przed ustaleniem wartości pól wyzerować strukturę (funkcją memset), aby pozostawione niezerowe wartości w nieznanych bliżej komponentach nie spowodowały odmiennej od zamierzonej interpretacji adresu.

Standard POSIX definiuje dwa obowiązkowe pola struktury sockaddr_un: całkowite *sun_family*, ustawiane na wartość *AF_LOCAL* oraz tablicę znakową sun_path. Tablica ta nie może mieć mniej niż 100 bajtów. W tablicy tej umieszczamy łańcuch zakończony zerem z nazwą pliku typu gniazdko (odpowiednik pliku nazwanego łącza). Zainicjowanie struktury może więc wyglądać następująco:

```
struct sockaddr_un addr;
...
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_LOCAL;
strcpy(addr.sun_path, "/tmp/gniazdko");
```

Dla struktury *sockaddr_in* standard przewiduje trzy pola obowiązkowe: całkowite *sin_family*, ustawiane na wartość *AF_INET*, *sin_addr* typu, który ze względów historycznych jest typem strukturalnym, a tak naprawdę jest 4-bajtową tablicą, zawierającą cztery oktety adresu IP oraz całkowite *sin_port*, które zawiera numer portu komunikacyjnego dla protokołów TCP i UDP. Numer ten jest umieszczany w strukturze w formacie zwanym *sieciowym* (w rzeczywistości jest to po prostu format *big-endian*, czyli od najstarszego bajtu). Plik netinet/in.h dostarcza dwóch funkcji: *htons* konwertującej format natywny do sieciowego (*host-to-network-short*) oraz odwrotna *ntohs* (*network-to-host-short*). Inicjalizacja takiej struktury może wyglądać tak:

```
struct sockaddr_in addr;
unsigned char ip[] = {192, 168, 0, 1};
  /* IP: 192.168.0.1 */
...
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_INET;
memcpy(&addr.sin_addr, ip, 4);
addr.sin_port = htons(80);
  /* port 80 */
```

Pozostawienie wartości zerowej w polu sin_addr oznacza „dowolny adres” (jest to stosowane w serwerach, do ustanawiania połączenia przez dowolny interfejs sieciowy), a w polu sin_port „port wybrany automatycznie” (stosowane w

klientach, aby przydzielić numer portu z puli adresów zwanych *efemerycznymi* czyli tymczasowymi).

6.3 Otwarcie bierne

Funkcja *socket* tworzy gniazdko, dostępne przez liczbę całkowitą, mającą charakter deskryptora plikowego. Funkcja pobiera trzy agrumenty: dziedzinę gniazdka (AF_LOCAL lub AF_INET), typ gniazdka (*SOCK_STREAM* dla gniazdek strumieniowych lub *SOCK_DGRAM* dla datagramowych) oraz protokół. Ten ostatni argument jest zwykle ustawiany na 0, co oznacza „dobierz protokół do typu gniazdka". W dziedzinie AF_INET typ SOCK_STREAM implikuje protokół TCP, SOCK_DGRAM – UDP.

Gniazdko po otwarciu biernym nie nadaje się do użycia bez związania (ang. *bind*) adresu. Czynią to funkcje *bind* oraz (tylko w klientach trybu strumieniowego) *connect*.

Poniższe podrozdziały opisują programowanie w dziedzinie AF_INET. Gniazdka w dziedzinie lokalnej zostaną omówione na końcu rozdziału.

6.4 Klient trybu strumieniowego

Funkcja *connect* ustanawia połączenie TCP przez gniazdko o deskryptorze podanym w pierwszym argumencie, z serwerem o adresie podanym w drugim. Trzeci argument to rozmiar struktury adresowej. Funkcja zwraca 0 przy udanym połączeniu.

Przykład 28 (tcpc1.c) *Program ustanawia połączenie z komputerem o adresie 127.0.0.1 (adres* loopback *czyli komputer klienta) na porcie TCP o numerze 10000. Po ustanowionym łączu sieciowym jest przesyłany napis "Hej", a następnie połączenie jest zrywane.*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
main()
{
    struct sockaddr_in server;
    unsigned char localhost[] = { 127, 0, 0, 1 };
    int sock;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    memcpy(&server.sin_addr, localhost, 4);
    server.sin_port = htons(10000);
    if (connect(sock, (struct sockaddr *) &server,
                sizeof(server))) {
        perror("connect");
        exit(1);
    }
    write(sock, "Hej\r\n", 5);
    close(sock);
    return 0;
}
```

Program uruchomiony bez żadnych wstępnych zabiegów, wypisze zapewne komunikat o braku powodzenia funkcji connect. Jest to spowodowane brakiem serwera TCP, nasłuchującego na porcie 10000. Możemy taki serwer uruchomić za pomocą programu `nc`. *Uruchomienie* `nc -l -p 10000` *uruchamia potrzebny serwer (tzw. „serwer echa"). Z konsoli, na której uruchomiliśmy serwer możemy teraz obserwować aktywność klienta.*

Zadanie 25 *Uruchom serwer WWW Apache poleceniem* `apachectl start`. *Dopisz do kodu klienta z poprzedniego przykładu następującą funkcjonalność: prześlij do serwera łańcuch* `''GET / HTTP/1.0\r\n\n''` *a następnie odczytaj (funkcją read) do bufora pamięciowego o rozmiarze 10000 znaków informację umieszczoną przez serwer w łączu gniazdkowym. Wypisz zawartość bufora na standardowe wyjście.*

6.5 Serwer trybu strumieniowego

Po utworzeniu gniazdka, serwer powinien *przywiązać* do niego adres sieciowy przy użyciu funkcji *bind*. Struktura adresowa przekazana do funkcji bind zawiera dane opisujące „naszą" stronę połączenia (w odróżnieniu od struktury z funkcji

connect). Pole sin_addr jest najczęściej nieinicjowane, co jest równoznaczne z tym, że serwer przyjmuje połączenia za pośrednictwem dowolnego z dostępnych interfejsów sieciowych. Pole sin_port zawiera z reguły numer portu TCP, na którym serwer będzie nasłuchiwał.

Funkcja bind zwraca błąd (wartość niezerową) najczęściej z jednej z dwóch przyczyn:

- podano numer portu, który jest już używany przez system (typowy komunikat: *address|port already in use*);
- numer portu jest mniejszy od 1024, a proces pracuje bez przywilejów administracyjnych (*permission denied*).

Ta pierwsza sytuacja zdarza się również wtedy, gdy proces–serwer zamknie już gniazdko, a nowy proces będzie próbował się do niego przywiązać bez odczekania charakterystycznego dla danego systemu czasu (zwykle kilku minut). Aby uniknąć tej sytuacji (szczególnie irytującej w momencie częstego uruchamiania procesu serwera), należy ustawić za pomocą funkcji *setsockopt* właściwość gniazdka, pozwalającą na dołączenie do niego używanego portu (porównaj następny przykład).

Po udanym przywiązaniu adresu, serwer powinien przeprowadzić gniazdko w *tryb nasłuchu* za pomocą funkcji *listen*. Jej drugi argument określa liczbę połączeń utrzymywanych w kolejce bez ich obsłużenia. Argument ten jest różnie interpretowany przez różne systemy, a zwyczajowo w programach używa się dość uniwersalnej wartości 5.

Odpowiednikiem funkcji connect po stronie serwera jest standardowo blokująca funkcja *accept*, która zwraca nowy deskryptor (tzw. deskryptor *gniazda połączonego* albo *transmisyjnego* w odróżnieniu od gniazda *nasłuchującego*, używanego jako pierwszy argument), za pomocą którego możemy transmitować dane od i do klienta. Drugim i trzecim argumentem funkcji accept są wskaźniki, które w najprostszym przypadku mogą być równe NULL. Jeżeli chcemy poznać strukturę połączonego klienta, podajemy w drugim argumencie adres zaalokowanej struktury sockaddr_in (zrzutowany na struct sockaddr *), a w trzecim adres liczby całkowitej typu *socklen_t*, do zapisania jej rozmiaru.

Gniazdko nasłuchujące jak i połączone powinno być niezależnie zamykane za pomocą funkcji close.

Przykład 29 (tcps1.c) *Poniższy program jest prostym iteracyjnym serwerem TCP, który nasłuchuje na porcie 10000, zczytując i wypisując na terminal dane*

przesyłane mu przez klientów. Odczytanie przez funkcję read zerowej liczby bajtów sygnalizuje rozłączenie się klienta (wartość ujemna sygnalizuje błąd).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFLEN 1000

main()
{
    struct sockaddr_in server;
    int sock, csock, opt, n;
    char buf[BUFLEN];
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    for (;;) {
        csock = accept(sock, NULL, NULL);
        printf("Klient polaczony\n");
        while ((n = read(csock, buf, BUFLEN - 1)) > 0) {
            buf[n] = 0;
            printf("Przeczytane bajty (%d): %s", n, buf);
        }
        close(csock);
        printf("Klient rozlaczony\n");
    }
}
```

Serwer możemy testować za pomocą klienta z przykładu 28, za pomocą polecenia `telnet localhost 10000` *czy* `nc localhost 10000`.

Zadanie 26 *Komentując wywołanie funkcji setsockopt w poprzednim przykładzie, ustal przybliżony czas braku możliwości ponownego przywiązania adresu w twoim systemie. Pamiętaj o wykonaniu choć jednego transferu danych przed zamknięciem serwera!*

Przykład 30 (tcps2.c) *Program ten jest trywialnym serwerem protokołu HTTP (WWW), nasłuchującym na porcie 10080.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFLEN 10000

main()
{
    struct sockaddr_in server, client;
    socklen_t csize;
    int sock, csock, opt, n;
    char buf[BUFLEN], response[BUFLEN];
    char header[] = "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n\n"
        "<html><body>\r\n"
        "Odpowiedz serwera na zadanie:<hr>\r\n"
        "<pre>\r\n",
        center[] = "</pre><hr>\r\n"
        "Adres klienta: <i>",
        footer[] = "</i>\r\n</body></html>\r\n";
    unsigned char *a;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
```

```
    server.sin_family = AF_INET;
    server.sin_port = htons(10080);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    for (;;) {
        csize = sizeof(client);
        csock =
            accept(sock, (struct sockaddr *) &client,
                   &csize);
        a = (unsigned char *) &client.sin_addr;
        n = read(csock, buf, BUFLEN - 1);
        buf[n] = 0;
        strcpy(response, header);
        strcat(response, buf);
        strcat(response, center);
        sprintf(response + strlen(response),
                "%d.%d.%d.%d:%d",
                a[0], a[1], a[2], a[3],
                ntohs(client.sin_port));
        strcat(response, footer);
        write(csock, response, strlen(response));
        close(csock);
    }
}
```

Na dowolne wezwanie od klienta (najlepiej przeglądarki internetowej, której podano jako adres połączeniowy http://127.0.0.1:10080*) serwer zwraca stronę internetową, zawierającą adres klienta oraz tekst żądania. Tekst znajdujący się na stronie pomiędzy poziomymi kreskami jest wysyłany przez przeglądarkę do serwera (powinna się tam znaleźć m.in. identyfikacja przeglądarki).*

Zadanie 27 *Dopisz do programu z przykładu 29 identyfikację adresu klienta.*

6.6 Klient i serwer trybu datagramowego

Cechą odróżniającą serwer UDP od serwera TCP jest prostota - oprócz wywołania funkcji bind kod serwera datagramowego nie robi nic więcej, zachowując pełną symetrię z działaniami klienta. Do wysyłania datagramów służy funkcja *sendto*, do ich odbierania standardowo blokująca funkcja *recvfrom*. Funkcje te mają trzy pierwsze argumenty analogiczne do funkcji write i read oraz trzy dodatkowe: flagi operacji (liczba całkowita, zwykle 0), wskaźnik na strukturę adresową adresata (sendto: wartości w niej powinny być odpowiednio zainicjowane przed wywołaniem) lub nadawcy (recvfrom zapisze tu odpowiednie dane) oraz długość (dla sendto, dla recvfrom wskaźnik na zmienną do przechowania długości) struktury adresowej. Obecność struktur adresowych w funkcjach transmisyjnych uświadamia nam brak sesji między klientem a serwerem.

Przykład 31 (udpc1.c/udps1.c) *Przedstawione niżej programy są możliwie prostymi reprezentantami klienta i serwera datagramowego. Klient wysyła co sekundę datagram zawierający kolejne liczby całkowite do serwera, uruchomionego na tym samym komputerze na porcie 10000.*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    struct sockaddr_in server;
    int sock, x, n;
    unsigned char localhost[] = { 127, 0, 0, 1 };
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    memcpy(&server.sin_addr, localhost, 4);
    server.sin_port = htons(10000);
    x = 0;
    for (;;) {
        n = sendto(sock, &x, sizeof(int), 0,
                   (struct sockaddr *) &server,
                   sizeof(server));
```

```
        printf("Wysłano bajtów: %d, wartość %d\n", n, x);
        x++;
        sleep(1);
    }
}
```

Serwer odczytuje liczby i wypisuje je na standardowe wyjście, wraz z adresem klienta, który je dostarczył.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    struct sockaddr_in server, client;
    socklen_t clen;
    int sock, x;
    unsigned char *a;
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    for (;;) {
        clen = sizeof(client);
        recvfrom(sock, &x, sizeof(int), 0,
                 (struct sockaddr *) &client, &clen);
        a = (unsigned char *) &client.sin_addr;
        printf
            ("Od %d.%d.%d.%d:%d otrzymano wartosc %d\n",
             a[0], a[1], a[2], a[3],
             ntohs(client.sin_port), x);
    }
}
```

Warto zwrócić uwagę na to, że bez względu na to czy serwer pracuje czy nie, klient dostaje od funkcji sendto wartość wskazującą na poprawne przesłanie danych. Demonstruje to (zamierzoną) zawodność protokołu UDP.

Zadanie 28 *Uzupełnij kod klienta i serwera o prostą sygnalizację sukcesu transmisji - niech serwer po odczytaniu wartości wyśle do klienta datagram „OK”. Klient nie wysyła następnej wartości bez uzyskania potwierdzenia.*

6.7 Serwery współbieżne

Przedstawione do tej pory serwery miały charakter iteracyjny - obsługiwały jednego klienta na raz (inaczej mówiąc: klienci obsługiwani byli po kolei). W wielu sytuacjach potrzebujemy serwera współbieżnego czyli takiego, który obsługuje wielu klientów równocześnie.

Przykład 32 (ctcpc1.c) *Dla testowania współbieżnych serwerów użyjemy specjalnie spreparowanego klienta, dokonującego wielu współbieżnych połączeń z serwerem TCP na porcie 10000 i zczytującego dostarczane przez niego dane. Program tworzy 5 wątków, wykonujących w pętli zestaw socket-connect-read-close. Współbieżnie wątek główny sygnalizuje nam na standardowym wyjściu upływający czas.*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>

#define BUFLEN 1024

void *loop(void *arg)
{
    struct sockaddr_in server;
    unsigned char localhost[] = { 127, 0, 0, 1 };
    char buf[BUFLEN];
    int sock, n, tid;
    tid = pthread_self();
    printf("Watek %d wystartowal\n", tid);
```

```
    for (;;) {
        sock = socket(AF_INET, SOCK_STREAM, 0);
        memset(&server, 0, sizeof(server));
        server.sin_family = AF_INET;
        memcpy(&server.sin_addr, localhost, 4);
        server.sin_port = htons(10000);
        if (connect(sock, (struct sockaddr *) &server,
                    sizeof(server))) {
            printf("Watek %d, polaczenie nieudane - "
                   "praca wstrzymana na 10 sekund\n", tid);
            close(sock);
            sleep(10);
            continue;
        }
        n = read(sock, buf, BUFLEN - 1);
        if (n > 0) {
            buf[n] = 0;
            printf
                ("Watek %d przeczytal %d bajtow, wartosc: %s",
                 tid, n, buf);
        }
        close(sock);
    }
}

main()
{
    int n;
    pthread_t tid;
    for (n = 0; n < 5; n++)
        pthread_create(&tid, NULL, loop, NULL);
    n = 0;
    for (;;) {
        printf("[%d]\n", n++);
        sleep(1);
    }
}
```

Każdy udany transfer jest odnotowywany na terminalu.

Punktem wyjścia będzie serwer iteracyjny, o wydzielonym w postaci osobnej funkcji kodzie obsługi klienta dla zwiększenia przejrzystości.

Przykład 33 (ctcps1.c) *Po połączeniu z klientem serwer losuje liczbę n z zakresu 5–10, zasypia na n sekund, po czym zapisuje do gniazdka łańcuch* `OK n sekund` *i zamyka gniazdko połączone. Cała akcja z gniazdem połączeniowym została wydzielona w funkcji* `service`*, o konstrukcji która ułatwi nam późniejszą rozbudowę - gniazdko jest przekazywane do funkcji przez adres zmiennej, zawierającej deskryptor.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

void *service(void *arg)
{
    int n;
    char buf[100];
    n = 5 + rand() % 6;
    sprintf(buf, "OK %d sekund\r\n", n);
    sleep(n);
    write(*((int *) arg), buf, strlen(buf));
    return arg;
}

main()
{
    struct sockaddr_in server;
    int sock, csock, opt;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
```

```
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    srand(time(NULL));
    for (;;) {
        csock = accept(sock, NULL, NULL);
        service(&csock);
        close(csock);
    }
}
```

Zwróć uwagę (testuj klientem z poprzedniego przykładu), że klienci są realizowani po kolei, w porządku FIFO.

Tradycyjnie najstarszym sposobem uzyskania współbieżności serwera (i nadal najczęściej stosowanym), jest konstrukcja *nowy klient = nowy proces*.

Przykład 34 (ctcps2.c) *Po wyjściu z funkcji accept, proces rozwidla się. W procesie macierzystym zamykamy gniazdko połączone, w potomnym – nasłuchowe. Proces potomny obsługuje klienta, macierzysty od razu nawraca do kolejnego wywołania accept. Do funkcji service jest przekazywany deskryptor gniazdka oraz liczba losowa (losowanie jest realizowane jedynie przez proces macierzysty, aby nie zaburzać procesu generowania wartości pseudolosowych).*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <sys/wait.h>

void handler_sigcld(int sig)
{
    int status;
```

```
    while (waitpid(-1, &status, WNOHANG) > 0);
}

struct svc_arg {
    int sock;
    int los;
};

void *service(void *arg)
{
    char buf[100];
    struct svc_arg *a = (struct svc_arg *) arg;
    sprintf(buf, "OK %d sekund\r\n", a->los);
    sleep(a->los);
    write(a->sock, buf, strlen(buf));
    return arg;
}

main()
{
    struct sockaddr_in server;
    int sock, opt;
    struct svc_arg a;
    signal(SIGCLD, handler_sigcld);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    srand(time(NULL));
    for (;;) {
        a.sock = accept(sock, NULL, NULL);
        a.los = 5 + rand() % 6;
```

```
        if (fork() == 0) {
            close(sock);
            service(&a);
            close(a.sock);
            exit(0);
        }
        close(a.sock);
    }
}
```

Uruchomienie tego serwera oraz klienta z przykładu 32 udowadnia, że wielu klientów jest obsługiwanych współbieżnie.

Jeżeli tylko system wspiera wielowątkowość, można pokusić się o realizację schematu *nowy klient = nowy watek*. Z reguły serwery wielowątkowe są wydajniejsze od wieloprocesowych (co oczywiście jest zauważalne przy wielu klientach łączących się prawie jednocześnie np. w obciążonych serwerach WWW).

Przykład 35 (ctcps3.c) *Wątek główny po wyjściu z funkcji accept tworzy nowy wątek, przekazując mu jako argument zaalokowaną strukturę* `svc_arg`. *Struktura ta musi być alokowana dla każdego wątku osobno (i przez ten wątek zwalniana), aby uniknąć jej zamazania przez kolejny tworzony wątek.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>

struct svc_arg {
    int sock;
    int los;
};

void *service(void *arg)
{
    char buf[100];
```

```
    struct svc_arg *a = (struct svc_arg *) arg;
    pthread_detach(pthread_self());
    sprintf(buf, "OK %d sekund\r\n", a->los);
    sleep(a->los);
    write(a->sock, buf, strlen(buf));
    close(a->sock);
    free(arg);
    return NULL;
}

main()
{
    struct sockaddr_in server;
    int sock, opt;
    struct svc_arg *a;
    pthread_t tid;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    srand(time(NULL));
    for (;;) {
        a = (struct svc_arg *)
            malloc(sizeof(struct svc_arg));
        a->sock = accept(sock, NULL, NULL);
        a->los = 5 + rand() % 6;
        pthread_create(&tid, NULL, service, a);
    }
}
```

Nowe wątki zmieniają swój stan na odłączony, aby uniknąć konieczności oczekiwania przez wątek główny na ich zakończenie.

Jeżeli wiele elementów aktywnych wykona równolegle operację accept na tym samym gniazdku, to w większości systemów unixowych w razie połączenia klienta budzi tylko jeden z nich, w porządku zbliżonym do rotacyjnego. Na tej zasadzie opiera się konstrukcja *wyprzedzającego* serwera TCP. Nowe procesy lub wątki tworzą się od razu w pewnej liczbie, tak aby obsłużyć klientów bez konieczności tworzenia nowego elementu aktywnego.

Przykład 36 (ctcps4.c) *Wątek główny tworzy 5 wątków dodatkowych, realizujących cykl accept-write-close.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>

int sock;

void *service(void *arg)
{
    char buf[100];
    int csock, los;
    pthread_detach(pthread_self());
    for (;;) {
        csock = accept(sock, NULL, NULL);
        los = 5 + rand() % 6;
        sprintf(buf, "OK(%d) %d sekund\r\n",
                pthread_self(), los);
        sleep(los);
        write(csock, buf, strlen(buf));
        close(csock);
    }
}

main()
{
    struct sockaddr_in server;
```

```
    int opt, n;
    pthread_t tid;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &opt, sizeof(opt));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(10000);
    if (bind(sock, (struct sockaddr *) &server,
             sizeof(server))) {
        perror("bind");
        exit(1);
    }
    listen(sock, 5);
    srand(time(NULL));
    for (n = 0; n < 5; n++)
        pthread_create(&tid, NULL, service, NULL);
    pause();
}
```

Aby udowodnić, że dane są produkowane przez różne wątki serwera, dodano do komunikatu numer wątku serwera.

Zadanie 29 *Napisz serwer wielowątkowy, który tworzy na starcie kilka wątków, ale funkcja accept jest wykonywana jedynie przez wątek główny. Otrzymany deskryptor jest zapisywany we wspólnej dla wszystkich wątków tablicy globalnej i odpowiedni wątek jest budzony do obsługi klienta.*

6.8 Superserwer inetd

Ponieważ obsługa gniazdka jest właściwie tożsama z jakimkolwiek innym deskryptorem plikowym, łatwo jest skonstruować program wykonujący za nas czynności związane z inicjalizacją serwera gniazdkowego i uruchamiający nowy proces z pliku binarnego dla każdego łączącego się klienta, proces w którym transmisja przez gniazdko jest zastąpiona transmisją przez ustalone numery deskryptorów np. 0 i 1 (standordowe wejście i wyjście). Takim właśnie programem jest *inetd* zwany też *superserwerem internetowym*.

Najłatwiej zrozumieć jak pracuje inetd analizując jego plik konfiguracyjny, standardowo `/etc/inetd.conf` (inny plik konfiguracyjny można wyspecyfikować w argumencie wywołania inetd). Każda nierozpoczęta znakiem # (hash) linia tego pliku definiuje akcję do wykonania w reakcji na połączenie klienta TCP lub UDP na określony port. Linia składa się z:

- nazwy lub numeru (niektóre implementacje inetd dopuszczają tylko nazwę) portu komunikacyjnego; wiązania nazwa–numer są ustalane poprzez plik `/etc/services`;
- typ gniazdka: *stream* odpowiada stałej SOCK_STREAM, *dgram* SOCK_DGRAM;
- protokół: *tcp* lub *udp*;
- współbieżność: *wait* oznacza serwer iteracyjny, *nowait* współbieżny;
- użytkownik–właściciel procesu serwera: pole to ma sens jedynie wtedy, gdy inetd jest uruchomiony z prawami administratora, bowiem tylko administrator może zmienić właściciela procesu; jeżeli uruchamiamy inetd bez praw administracyjnych powinniśmy podać tu swój identyfikator;
- ciąg argumentów funkcji execve.

Jeżeli na odpowiedni port zgłosi się klient, deskryptor gniazdka zostaje zrzutowany (za pomocą funkcji *dup2*) na deskryptory standardowego wejścia i wyjścia, następuje rozwidlenie procesu, zmiana identyfikatora użytkownika oraz odpowiednie wywołanie exec.

Przykład 37 (inetd1.conf/inetd1.c) *Niech plik* `inetd1.conf` *zawiera jedną linię:*

```
10000 stream tcp nowait knoppix /tmp/inetd1 inetd1
```

Umieśćmy kompilat programu:

```
#include <unistd.h>

main()
{
    char buf[1000];
    int n;
```

```
    write(1, "Wprowadz lancuch znakow\r\n", 25);
    n = read(0, buf, 1000);
    write(1, "Wprowadziles: ", 14);
    write(1, buf, n);
    return 0;
}
```

w katalogu `/tmp` *i wydajmy polecenie:*

```
/usr/sbin/inetd -d inetd1.conf
```

Opcja -d powoduje brak przejścia programu w tryb demona i wypisywanie komunikatów diagnostycznych na terminalu. Sprawdźmy działanie serwera na porcie TCP numer 10000 poleceniem `telnet localhost 10000`.

Warto zauważyć, że kod programu dostarczającego usługi jest całkowicie pozbawiony funkcji gniazdkowych.

Zadanie 30 *Przerób kod serwera HTTP z przykładu 30 tak, aby uruchamiać go za pośrednictwem inetd.*

6.9 Usługi nazewnicze

Korzystanie z numerów IP jest niewygodne. Jesteśmy przyzwyczajeni do używania dla komputerów nazw symbolicznych. Może być wielu dostawców takiej translacji - lokalna konfiguracja komputera (plik `/etc/hosts`), system NIS czy DNS. Nie musimy wnikać w to, co dostarczyło nam informacji - tzw. *resolver* czyli element systemu odpowiedzialny za sposób ustalania nazw sieciowych jest zapewne poprawnie skonfigurowany i programistom wystarczy wywołać odpowiednie funkcje.

Funkcja *gethostbyname* (netdb.h) zwraca wskaźnik na statyczną strukturę typu `struct hostent`, która zawiera między innymi:

- w polu *h_name* wskaźnik na oficjalną (kanoniczną) nazwę komputera;
- w polu *h_addr_list* tablicę wskaźników na adresy IP komputera; tablica ta zakończona jest wskaźnikiem pustym; wskaźniki są typu `char *`, a wskazywane bajty są ułożone w kolejności sieciowej.

Przykład 38 (name1.c) *Program wczytuje ze standardowego wejścia nazwy komputerów, wypisując ich adresy IP (tylko pierwsze pozycje z listy). Zwróć uwagę na sposób sygnalizowania błędnej nazwy.*

```
#include <stdio.h>
#include <netdb.h>

main()
{
    struct hostent *h;
    char s[200];
    unsigned char *a;
    for (;;) {
        scanf("%s", s);
        h = gethostbyname(s);
        if (h) {
            a = (unsigned char *) h->h_addr_list[0];
            if (a)
                printf("%d.%d.%d.%d\n",
                       a[0], a[1], a[2], a[3]);
            else
                printf("brak nazwy\n");
        } else
            perror("gethostbyname");
    }
}
```

Przy braku dostępu do internetu testuj program zmieniając z poziomu administratora plik `/etc/hosts`.

Funkcja *gethostbyaddr* zwraca wskaźnik do struktury hostent na podstawie podanego w pierwszym argumencie wskaźnika (typu `char *`) na obszar zawierający binarny adres IP w formacie sieciowym. Drugim argumentem jest rozmiar adresu (w internecie 4), trzecim - rodzina protokołów (AF_INET).

Zadanie 31 *Napisz program, który na podstawie czterech liczb–oktetów wprowadzonych ze standardowego wejścia, wypisze nazwę symboliczną odpowiedniego komputera.*

6.10 Gniazdka w dziedzinie lokalnej

Inicjalizacja struktury adresowej sockaddr_un polega na przekopiowaniu do pola *sun_path* ścieżki dostępu do pliku specjalnego typu gniazdko. Plik ten jest tworzony przez funkcję bind, o ile nie istnieje. Niestety, prawa dostępu do utworzonego pliku specjalnego zależą od systemu. Przez utworzone gniazdko można dokonywać transferu danych analogicznego do gniazd internetowych, otrzymując funkcjonalność podobną do łącz nazwanych.

Przykład 39 (sock1.c) *Program jest serwerem, produkującym w pętli do każdego klienta przyłączonego przez gniazdko lokalne* `/tmp/gniazdko` *napis „OK”. Jeżeli wcześniej istniał plik* `/tmp/gniazdko` *jest kasowany.*

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    struct sockaddr_un server;
    int sock, csock;
    sock = socket(AF_LOCAL, SOCK_STREAM, 0);
    memset(&server, 0, sizeof(server));
    server.sun_family = AF_LOCAL;
    strcpy(server.sun_path, "/tmp/gniazdko");
    unlink(server.sun_path);
    bind(sock, (struct sockaddr *) &server, sizeof(server));
    listen(sock, 5);
    for (;;) {
        csock = accept(sock, NULL, NULL);
        write(csock, "OK\r\n", 4);
        close(csock);
    }
}
```

Zadanie 32 *Napisz klienta dla serwera z poprzedniego przykładu.*

Odpowiednikiem łącz nienazwanych są gniazdka utworzone za pomocą funkcji *socketpair*. Parametry tej funkcji są takie jak funkcji socket plus dwuelemen-

towa tablica tak jak w funkcji pipe. Po skutecznym wywołaniu w tablicy tej mamy deskryptory dwóch połączonych, nienazwanych gniazdek lokalnych.

W odróżnieniu od deskryptorów zwracanych przez funkcję pipe, komunikacja między deskryptorami utworzonymi przez socketpair jest dwukierunkowa. Do transferu danych używamy funkcji read i write (bez bind, connect itp.), tak jak w łączach nienazwanych.

Zadanie 33 *Przystosuj program z przykładu 12 do komunikacji za pomocą nienazwanych gniazd lokalnych.*

6.11 Zwielokrotnianie

Technika zwana *zwielokrotnianiem* (ang. *multiplexing*) ma zastosowanie do dowolnych deskryptorów plikowych, ale najczęściej i najbardziej efektywnie jest stosowana z gniazdkami.

Na potrzeby funkcji zwielokratniającej *select* (sys/select.h) definiuje się typ `fd_set`, którego zmienne definiują zbiór deskryptorów (szczegółowa implementacja tego typu nie jest istotna). Plik sys/select.h dostarcza zestawu makr do wykonywania operacji na zbiorze deskryptorów. Jeżeli `ds` jest zmienną typu `fd_set`, a `d` pojedynczym deskryptorem to:

- `FD_ZERO(&ds)` czyni zbiór deskryptorów pustym;
- `FD_SET(d, &ds)` dołącza do zbioru podany deskryptor;
- `FD_CLR(d, &ds)` usuwa ze zbioru podany deskryptor;
- `FD_ISSET(d, &ds)` zwraca wartość niezerową, o ile deskryptor jest w zbiorze.

Funkcja select pobiera w argumentach m.in. trzy wskaźniki na zbiory deskryptorów: zbiór deskryptorów *gotowych do czytania*, *gotowych do pisania* oraz zbiór deskryptorów, dla których powstała *sytuacja wyjątkowa*. Tymi ostatnimi zajmować się nie będziemy.

Deskryptor jest gotowy do czytania, gdy:

- pojawiły się na nim nowe dane (read nie zablokuje procesu);
- został zamknięty po drugiej stronie łącza (np. przez klienta gniazdkowego - read nie zablokuje procesu zwracając 0 czyli koniec połączenia);

- dla gniazdka nasłuchującego - nawiązano połączenie (accept nie zablokuje procesu);
- wystąpił nieobsłużony błąd na deskryptorze (read zwróci -1).

Deskryptor jest gotowy do pisania, gdy:

- po drugiej stronie łącza pojawiło się zapotrzebowanie na dane;
- został zamknięty po drugiej stronie łącza;
- wystąpił błąd.

Funkcja select pobiera 5 argumentów wywołania:

- największy numer deskryptora do zbadania plus 1; motywacją wprowadzenia tego argumentu jest chęć optymalizacji kodu select; nie da się jednak ukryć, że komplikuje to mocno przygotowanie argumentów;
- trzy wskaźniki na zbiory deskryptorów: gotowych do czytania, pisania i takich dla których wystąpiły sytuacje wyjątkowe; na wejściu zbiory te powinny zawierać te deskryptory, które nas interesują; na wyjściu zbiory te będą zawierać te deskryptory, które są „gotowe”;
- wskaźnik na strukturę typu `struct timeval` (sys/time.h), definiującą czas oczekiwania (struktura ta składa się z pól `tv_sec` zawierającego sekundy oraz `tv_usec` - mikrosekundy; rzeczywista rozdzielczość jest zwykle dużo mniejsza, najczęściej są to dziesiąte sekundy); wartość NULL oznacza nieskończone blokowanie (najczęściej stosowane), wartość `{0, 0}` oznacza brak blokowania i natychmiastowy powrót po sprawdzeniu zbiorów.

Funkcja select zwraca liczbę gotowych deskryptorów lub 0 przy upłynięciu czasu oczekiwania oraz -1 w przypadku błędu.

Przykład 40 (sel1.c) *Program jest serwerem, nasłuchującym na pięciu portach TCP, 10001–10005, pozwalającym dodatkowo na wprowadzanie danych ze standardowego wejścia. Wszystkie dane są traktowane tak samo: serwer wypisuje je na standardowe wyjście wraz z informacją skąd pochodzą.*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define NPORTS 5
#define BUFLEN 1000

main()
{
    int sock[NPORTS];
    struct sockaddr_in server[NPORTS];
    fd_set ds;
    int n, d, opt, maxds, csock;
    char buf[BUFLEN];
    maxds = 0;
    for (n = 0; n < NPORTS; n++) {
        sock[n] = socket(AF_INET, SOCK_STREAM, 0);
        if (sock[n] > maxds)
            maxds = sock[n];
        opt = 1;
        setsockopt(sock[n], SOL_SOCKET, SO_REUSEADDR,
                   &opt, sizeof(opt));
        memset(&server[n], 0, sizeof(server[n]));
        server[n].sin_family = AF_INET;
        server[n].sin_port = htons(10001 + n);
        if (bind
            (sock[n], (struct sockaddr *) &server[n],
             sizeof(server[n]))) {
            perror("bind");
            exit(1);
        }
        listen(sock[n], 5);
    }
    for (;;) {
        FD_ZERO(&ds);
        FD_SET(0, &ds);
        for (n = 0; n < NPORTS; n++)
            FD_SET(sock[n], &ds);
```

```
        select(maxds + 1, &ds, NULL, NULL, NULL);
        for (d = 0; d <= maxds; d++) {
            if (FD_ISSET(d, &ds)) {
                csock = d > 0 ? accept(d, NULL, NULL) : 0;
                n = read(csock, buf, BUFLEN - 1);
                if (n > 0) {
                    buf[n] = 0;
                    printf
                        ("Z deskryptora %d odczytano: %s",
                         csock, buf);
                }
                if (csock > 0)
                    close(csock);
            }
        }
    }
}
```

Sprawdź program wprowadzając naprzemiennie łańcuchy ze standardowego wejścia oraz łącząc się programem telnet z portami 10001–10005.

Zadanie 34 *Zastąp w programie z poprzedniego przykładu komunikaty „z deskryptora nr” komunikatami „ze standardowego wejścia” lub „z portu TCP nr”.*

Zadanie 35 *Napisz funkcjonalny odpowiednik programu z poprzedniego przykładu, używający wielowątkowości.*

Rozdział 7

Zdalne wywołanie procedury

7.1 Podstawy

Typowa komunikacja między klientem a serwerem sieciowym odbywa się w modelu *wyślij zapytanie–odbierz odpowiedź*. Przenosząc ten model do programowania, zauważamy analogię do wywołania funkcji: zapytanie to jej argumenty, wartość zwrotna to odpowiedź. Jeżeli nie oczekujemy odpowiedzi od serwera, to mamy do czynienia z funkcją o typie zwrotnym `void`, jeżeli chcemy jedynie zczytać jakieś dane z serwera - funkcją bezargumentową. Jeżeli wywołujemy funkcję, to przestajemy wykonywać liniowo nasz kod - oddajemy sterowanie „gdzieś indziej", do „obcego" często fragmentu programu. W języku sieciowym: blokujemy klienta do czasu nadejścia odpowiedzi od serwera.

Fundamentem zdalnego wywołania procedury czyli RPC (ang. *remote procedure call*) jest ukrycie komunikacji sieciowej w języku programowania tak, aby uruchomienie kodu na innej maszynie odbywało się tak, jak wywołanie lokalnej funkcji. Zwykle odbywa się to w sposób następujący:

- programista serwera pisze *interfejs serwera*, zawierający specyfikację tego, co serwer ma do zaproponowania i sposobu, w jaki należy się tego dostać;
- na podstawie interfejsu, programista konstruuje implementację usług serwerowych; zwykle wytwarza na bazie interfejsu specjalnym narzędziem tzw. „pieniek serwera" (ang. *server stub*) czyli zbiór niezbędnych funkcji do zarządzania połączeniami klienckim; następnie tworzy implementację funkcji interfejsu;
- programista klienta, wytwarza specjalnym narzędziem na podstawie upublicznionego interfejsu serwera tzw. „pieniek klienta" (ang. *client stub*),

zawierający kod funkcji paralelnych do funkcji serwerowych, które zamiast odpowiedniej funkcjonalności zapewniają połączenie z serwerem, przekazanie argumentów i zwrot wyników.

Na ogół RPC usuwają problem heterogeniczności architektur klienta i serwera. W pieńkach instaluje się procedury konwersji typów, tak że programista nie musi być świadom ich różnej reprezentacji na obu maszynach.

Programowanie przy użyciu RPC jest na tyle łatwe, że często stosuje się tę technologię również do komunikacji w obrębie tej samej maszyny.

Omówimy tu najpopularniejszą implementację RPC, tzw. *Sun RPC* (od nazwy firmy), dostępną w wielu systemach unixowych.

7.2 Język opisu interfejsu

Język opisu interfejsu w Sun RPC ma charakter deklaratywny - nie zawiera definicji funkcji, a jedynie ich nagłówkowe deklaracje oraz definicje typów i stałych. Jest w swojej budowie podobny do języka C, z pewnymi uproszczeniami.

Przykład 41 (rpc1.x) *Oto przykładowy interfejs serwera, dostarczającego dwóch funkcji:* dodaj *do dodawania dwóch liczb rzeczywistych oraz* pierwiastek *do uzyskiwania pierwiastka kwadratowego argumentu.*

```
struct double2 {
  double x;
  double y;
};

program RPC1 {
  version RPC1VER {
    double dodaj(double2) = 1;
    double pierwiastek(double) = 2;
  } = 1;
} = 0x20000001;
```

Definicję interfejsu rozpoczyna deklaracja struktury `double2`*, o dwóch komponentach typu* `double`*. Struktura ta będzie nam potrzebna, aby sprostać wymaganiom języka - funkcje serwera muszą być jednoargumentowe (nowsze implementacje nie zawierają tego ograniczenia).*

Blok `program` *deklaruje serwer o nazwie* `RPC1` *i numerze* `0x30000001`. *W ramach serwera będzie dostępna jedna wersja zbioru funkcji,* `RPC1VER` *o wartości* `1`. *Zbiór ten składa się z dwóch funkcji: pobierającej dwie liczby* `double` *i zwracającej* `double` *o nazwie* `dodaj` *i numerze porządkowym 1 oraz* `pierwiastek` *o oczywistej konstrukcji.*

Numeracje bloków rządzą się następującymi prawami:

- numer programu jest 32–bitową liczbą całkowitą; zakres 0x0–0x1FFFFFFF jest zdefiniowany przez pakiet Sun RPC (w pewnym sensie jest odpowiednikiem zakresu *well–known ports* TCP/UDP czyli portów 1–1023); zakres 0x20000000–0x3FFFFFFF jest przeznaczony dla programistów; zakres 0x40000000–0x5FFFFFFF to tymczasowe usługi RPC (odpowiednik portów efemerycznych TCP/UDP); numery powyżej 0x5FFFFFFF są zarezerwowane dla przyszłych zastosowań; numer powinien być unikalny w ramach systemu;
- numer wersji jest 32–bitową liczbą całkowitą większą od zera, unikalną w ramach programu;
- numer funkcji jest 32–bitową liczbą całkowitą większą od zera, unikalną w ramach wersji.

Język opisu interfejsu rozpoznaje m.in. następujące typy danych:

- typ pusty `void`;
- całkowite `char`, `short`, `int`, `long`, z ewentualnym modyfikatorem `unsigned`;
- zmiennoprzecinkowe `float` i `double`;
- tablice o stałej (`typ nazwa[liczba]`) i zmiennej długości (`typ nazwa<liczba>`; podanie maksymalnego rozmiaru może być pominięte); tablice o zmiennej długości są mapowane w języku C na struktury, zawierające długość i sekwencję odpowiednich wartości;
- `string nazwa<liczba>` dla przechowywania łańcuchów znakowych; liczba może być opuszczona (brak limitu długości);
- struktury.

Dostępna jest znana z języka C dyrektywa `typedef`. Jej użycie jest konieczne z typami tablicowymi.

Przykład 42 (rpc2.x) *Interfejs serwera autoryzującego może wyglądać następująco:*

```
typedef string str8<8>;

struct authdata {
  str8 user;
  string password<>;
};

program AUTHSERV {
  version AUTHVER {
    int login(authdata) = 1;
    void logout(void) = 2;
    str8 loginname(void) = 3;
  } = 1;
} = 0x20000002;
```

Definicje interfejsów serwerów RPC umieszczamy zwyczajowo w plikach z rozszerzeniem `.x`.

7.3 Prekompilator rpcgen

Program *rpcgen* służy do generowania pieńków serwera oraz klienta. Wywołany z nazwą pliku interfejsu, powiedzmy `nazwa.x`, generuje przynajmniej trzy pliki źródłowe w języku C:

- `nazwa.h`, plik nagłówkowy z definicjami stałych, typów oraz z prototypami funkcji serwerowych i klienckich;
- `nazwa_svc`, pieniek serwera;
- `nazwa_clnt.c`, pieniek klienta.

Często dodatkowym efektem działania rpcgen jest plik `nazwa_xdr.c`, zawierający funkcje konwertujące typy z i do formatu wspólnego (*XDR* od *external data representation*).

W zasadzie programista nie musi oglądać ani tym bardziej zmieniać wygenerowanych przez rpcgen plików. Będą one składnikami programu serwera i klienta, ale nie mogą stanowić same gotowych programów, brakuje bowiem:

- implementacji funkcji interfejsu w serwerze;
- funkcji głównej (`main`) w kliencie.

Pliki z tymi uzupełnieniami powinny zostać dodane przez programistę i skompilowane wspólnie z kodem wygenerowanym przez rpcgen.

Przykład 43 (rpc1.x) *Po przyłożeniu prekompilatora rpcgen do pliku z przykładu 41 (środowisko: linux, gcc) dostajemy cztery pliki: rpc1.h, rpc1_xdr.c, rpc1_svc.c oraz rpc1_clnt.c. W pliku nagłówkowym znajdziemy m.in.:*

- *definicję struktury* `double2`*, analogiczną do tej z interfejsu;*
- *ustalenie równoważnika typów* `double2` = `struct double2`*;*
- *definicję stałych* `RPC1` *i* `RPC1VER`*, równych odpowiednim wartościom z pliku interfejsu (odpowiednio 0x20000001 i 1);*
- *prototypy funkcji serwerowych,* `dodaj_1_svc` *i* `pierwiastek_1_svc`*;*
- *prototypy funkcji klienckich,* `dodaj_1` *i* `pierwiastek_1`*.*

Prototypy funkcji serwerowych i klienckich są różne od tych z pliku interfejsu:

- do nazw funkcji serwerowych dodawany jest przyrostek składający się ze znaku _, numeru wersji i łańcucha `_svc`;
- do nazw funkcji klienckich dodawany jest przyrostek składający się ze znaku _ i numeru wersji;
- argumenty oraz wartości zwrotne zamieniają się na wskaźniki na odpowiednie typy (np. `double` na `double *`) oraz podlegają dostosowaniu do możliwości języka C (np. typ `string` zostaje zamieniony na `char *`);
- dodatkowym drugim argumentem funkcji serwerowych staje się wskaźnik na strukturę typu `svc_req`, tzw. strukturę wywołania;
- dodatkowym drugim argumentem funkcji klienckich staje się wskaźnik na strukturę typu `CLIENT`, tzw. strukturę klienta.

7.4 Serwer RPC

Do skompletowania kodu serwera RPC należy zaimplementować ciała funkcji serwerowych.

Przykład 44 (rpc1_server.c) *Poniższy plik stanowi definicję funkcji serwera z przykładu 41.*

```
#include "rpc1.h"
#include <math.h>

double *dodaj_1_svc(double2 * arg, struct svc_req *req)
{
    static double wynik;
    wynik = arg->x + arg->y;
    return &wynik;
}

double *pierwiastek_1_svc(double *arg, struct svc_req *req)
{
    static double wynik;
    wynik = (*arg >= 0) ? sqrt(*arg) : -1;
    return &wynik;
}
```

Całość kompilujemy (linux, gcc) poleceniem:

```
cc rpc1_server.c rpc1_svc.c rpc1_xdr.c -lm
```

Opcja `-lm` *została użyta ze względu na funkcję biblioteczną* `sqrt` *(dołączenie biblioteki matematycznej).*

Dwie cechy implementowania funkcji serwerowych mogą stanowić kłopot dla początkującego programisty:

- ponieważ rezultaty są przekazywane przez wskaźniki, należy albo zaalokować miejsce na rezultat przed jego obliczeniem (np. funkcją malloc) albo umieścić rezultat w zmiennej statycznej i zwrócić jej adres;

- nie należy dopuścić do błędów, które mogą prowadzić do załamania serwera, który z natury jest „wielorazowy”; w przypadku detekcji sytuacji grożącej załamaniem serwera należy powrócić z funkcji serwerowej z wartością NULL, która oznaczać będzie dla klienta niepowodzenie zdalnej procedury.

W przypadku stosowania tablic lub łancuchów znakowych pierwsza trudność się potęguje ze względu na mapowanie tych obiektów na wskaźniki języka C.

Przykład 45 (rpc2_server.c) *Przedstawmy prostą implementację serwera z przykładu 42.*

```
#include "rpc2.h"
#include <string.h>

static char _loginname[8 + 1] = "";

int *login_1_svc(authdata *arg, struct svc_req *req)
{
    static int wynik;
    if (strlen(_loginname) > 0)
        wynik = 0;              /* najpierw logout! */
    else {
        strcpy(_loginname, arg->user);
        wynik = 1;              /* kazde haslo ok... */
    }
    return &wynik;
}

void *logout_1_svc(void *arg, struct svc_req *req)
{
    _loginname[0] = 0;
    return _loginname;
}

str8 *loginname_1_svc(void *arg, struct svc_req *req)
{
    static str8 wynik;          /* wskaznik */
    wynik = _loginname;
    return &wynik;              /* wskaznik na wskaznik */
}
```

Warto zwrócić uwagę, że nawet funkcja zadeklarowana w interfejsie jako zwracająca `void` *(logout) powinna zwracać w przypadku sukcesu jakiś niezerowy wskaźnik.*

Uruchomienie serwera RPC nie da żadnych widocznych na terminalu efektów - program zablokuje się w oczekiwaniu na zgłaszających się klientów.

7.5 Klient RPC

Programista który chce skorzystać z funkcji oferowanych przez serwer RPC, powinien znać:

- interfejs serwera w postaci odpowiedniego pliku `.x`;
- nazwę sieciową komputera serwera.

Z interfejsu serwera za pomocą rpcgen programista wytwarza pieńki i dopisuje własny kod, zawierający m.in. funkcję main.

Przykład 46 (rpc1_client.c) *Program jest prostym klientem serwera z przykładu 41. Dokonuje dodania dwóch liczb, jednego udanego pierwiastkowania i jednego nieudanego podejścia do pierwiastkowania liczby ujemnej (raportowanego przez wartość zwrotną -1).*

```
#include "rpc1.h"
#include <stdio.h>
#include <stdlib.h>

#define SVCHOST "localhost"

main()
{
    double2 arg2;
    double arg;
    double *wynik;
    CLIENT *cl;
    cl = clnt_create(SVCHOST, RPC1, RPC1VER, "tcp");
    if (!cl) {
        clnt_pcreateerror(SVCHOST);
```

```
        exit(1);
    }
    arg2.x = 1.5;
    arg2.y = 2.7;
    wynik = dodaj_1(&arg2, cl);
    if (wynik) {
        printf("%lg + %lg = %lg\n", arg2.x, arg2.y, *wynik);
        clnt_freeres(cl, (xdrproc_t) xdr_double,
                     (char *) wynik);
    } else
        clnt_perror(cl, "dodaj_1");
    arg = 81;
    wynik = pierwiastek_1(&arg, cl);
    if (wynik) {
        printf("sqrt(%lg) = %lg\n", arg, *wynik);
        clnt_freeres(cl, (xdrproc_t) xdr_double,
                     (char *) wynik);
    } else
        clnt_perror(cl, "pierwiastek_1");
    arg = -81;
    wynik = pierwiastek_1(&arg, cl);
    if (wynik) {
        printf("sqrt(%lg) = %lg\n", arg, *wynik);
        clnt_freeres(cl, (xdrproc_t) xdr_double,
                     (char *) wynik);
    } else
        clnt_perror(cl, "pierwiastek_1");
    clnt_destroy(cl);
    return 0;
}
```

Całość kompilujemy (linux, gcc) poleceniem:

```
cc rpc1_client.c rpc1_clnt.c rpc1_xdr.c
```

Dla przetestowania klienta powinniśmy najpierw uruchomić na tej samej maszynie serwer.

Klient nawiązuje logiczne połączenie z serwerem dzięki funkcji *clnt_create* (o prototypie z pliku rpc/rpc.h, który jest włączany przez wygenerowany plik nagłówkowy), podając jej jako argumenty: nazwę sieciową komputera na którym

pracuje serwer, numer programu (najczęściej poprzez stałą z wygenerowanego pliku nagłówkowego), numer wersji oraz łańcuch definiujący protokół transportowy. Jest to jedyne miejsce w programowaniu RPC, gdzie ujawniamy pracę sieciową.

Rezultatem funkcji clnt_create jest wskaźnik na strukturę typu *CLIENT*, która będzie w następnych krokach naszym identyfikatorem sesji RPC (będzie m.in. drugim argumentem wywołań funkcji zdalnych). Wartość zwrotna NULL oznacza błąd połączenia, wynikający z nieosiągalności komputera serwera, odpowiednich usług uruchomionych na tym komputerze, braku uruchomionego programu serwera bądź niepoprawnym określeniu numerów programu, wersji czy protokołu. Funkcja *clnt_pcreateerror* jest odpowiednikiem funkcji perror, pozwalając nam na diagnozę błędu.

Brak powodzenia wywołania funkcji zdalnej powoduje zwrot przez funkcje pieńka wartości NULL. Diagnozę błędu ułatwia funkcja *clnt_perror*.

Funkcja *clnt_freeres* zwalnia pamięć przydzieloną dla rezultatów funkcji zdalnej. Jej drugi argument jest wskaźnikiem na odpowiednią funkcję konwertującą w bibliotece XDR (o nazwie *xdr_typwartoscizwracanej*; dla typów własnych programisty, prototypy odpowiednich funkcji konwertujących znajdują się w wygenerowanym pliku nagłówkowym). Argumenty funkcji clnt_freeres powinny być odpowiednio zrzutowane.

Funkcja *clnt_destroy* zamyka sesję RPC.

Zadanie 36 *Zwróć uwagę, że błąd w argumentach funkcji* `pierwiastek_1_svc` *w programie serwera (przykład 44) nie został obsłużony w, wydawałoby się, najbardziej oczywisty sposób, a mianowicie przez zwrócenie wartości NULL. Zbadaj zachowanie klienta przy zwróceniu NULL przez implementację funkcji serwerowej.*

Zadanie 37 *Dopisz do funkcjonalności serwera funkcję dzielenia dwóch liczb. Jak rozwiążesz problem dzielenia przez zero?*

Podawanie argumentów funkcji zdalnych i odbiór wyników przez wskaźniki często komplikuje programowanie, co widać szczególnie w pracy z tablicami.

Przykład 47 (rpc2_client.c) *Przykładowy program klienta dla serwera z przykładu 42 nie zawiera wywołań żadnych dodatkowych funkcji kontrolnych.*

```
#include "rpc2.h"
```

```
#include <stdio.h>

main()
{
    CLIENT *cl;
    str8 *logname;
    authdata a = { "kowalski", "KazdeHasloJestDobre" };
    cl = clnt_create("localhost", AUTHSERV, AUTHVER, "tcp");
    logname = loginname_1(NULL, cl);
    printf("Zalogowany uzytkownik: %s\n", *logname);
    if (login_1(&a, cl)) {
        logname = loginname_1(NULL, cl);
        printf("Zalogowany uzytkownik: %s\n", *logname);
    } else
        printf("Logowanie nieudane\n");
    logout_1(NULL, cl);
    logname = loginname_1(NULL, cl);
    printf("Zalogowany uzytkownik: %s\n", *logname);
    return 0;
}
```

Zadanie 38 *W poprzednim przykładzie zamknij kod po wywołaniu clnt_create w pętli nieskończonej. Po uruchomieniu klienta śledź (np. programem* `top`*) wykorzystanie pamięci przez proces klienta. Co obserwujesz? Zaradź problemowi.*

Zadanie 39 *Napisz za pomocą technologii RPC prosty* serwer plików, *udostępniający jedną funkcję o interfejsie:*

```
typedef char memblock<>;
typedef string filename<>;
...
    memblock getfile(filename);
```

Zwróć uwagę na mapowanie typu `memblock` *zdefiniowane w wygenerowanym pliku nagłówkowym.*

7.6 Limit czasu wywołania

Wywołanie funkcji zdalnej jest czasowo blokujące. Jeżeli serwer nie prześle odpowiedzi w odpowiednim czasie (funkcja serwerowa nie zakończy się zwróce-

niem niepustego wskaźnika), klient dostanie z pieńka wartość NULL.

Przykład 48 (rpc3.x/rpc3_server.c/rpc3_client.c) *Serwer udostępnia jedną funkcję,* `rsleep`*, która blokuje się na zadaną liczbę sekund. Plik interfejsu może wyglądać następująco:*

```
program RPC3 {
  version RPC3_VER {
    void rsleep(int) = 1;
  } = 1;
} = 0x30000001;
```

Jedyna funkcja serwerowa ma ciało:

```
#include "rpc3.h"
#include <unistd.h>
#include <stdio.h>

void *rsleep_1_svc(int *sec, struct svc_req *req)
{
    static char foo;
    printf("Zamówiono:    rsleep(%d)\n", *sec);
    sleep(*sec);
    printf("Zrealizowano: rsleep(%d)\n", *sec);
    return &foo;
}
```

Kod testowego klienta przedstawiony jest poniżej:

```
#include "rpc3.h"
#include <stdio.h>

CLIENT *cl;

void remote_sleep(int sec)
{
    if (rsleep_1(&sec, cl))
        printf("rsleep(%d) ok\n", sec);
    else {
```

```
        fprintf(stderr, "rsleep(%d) error", sec);
        clnt_perror(cl, "clnt_call");
    }
}

main()
{
    cl = clnt_create("localhost", RPC3, RPC3_VER, "tcp");
    remote_sleep(5);
    remote_sleep(10);
    remote_sleep(40);
    return 0;
}
```

Serwer dla większej kontroli pisze na swojej konsoli komunikaty o rozpoczęciu i zakończeniu funkcji serwerowej. Klientowi udaje się zrealizować zamówienie na „zdalną drzemkę" 5–cio i 10–sekundową. Zamówienie 40–sekundowe pozostaje niezrealizowane - po 30 sekundach klient dostaje NULL, choć serwer kontynuuje zamówioną operację.

Domyślny czas blokowania funkcji zdalnej wynosi 30 sekund. Można go zmienić za pomocą funkcji `clnt_control`.

Przykład 49 (rpc3a_client.c) *Kod klienta z poprzedniego przykładu zmodyfikowano tak, aby zamówienie trwające 40 sekund mogło być wykonane synchronicznie.*

```
#include "rpc3.h"
#include <stdio.h>
#include <sys/time.h>

CLIENT *cl;

void remote_sleep(int sec)
{
    if (rsleep_1(&sec, cl))
        printf("rsleep(%d) ok\n", sec);
    else {
        fprintf(stderr, "rsleep(%d) error", sec);
        clnt_perror(cl, "clnt_call");
```

```
    }
}

main()
{
    struct timeval tv;
    cl = clnt_create("localhost", RPC3, RPC3_VER, "tcp");
    tv.tv_sec = 60; tv.tv_usec = 0;
    clnt_control(cl, CLSET_TIMEOUT, (char *) &tv);
    remote_sleep(5);
    remote_sleep(10);
    remote_sleep(40);
    return 0;
}
```

Niekiedy możliwość wpływania na czas blokowania wykorzystuje się do realizacji techniki *fire–and–forget*: klient ustawia czas blokowania na 0 i wywołuje funkcję serwerową, nie dbając o jej rezultat. Mimo że klient dostaje błąd (NULL), serwer wykonuje zamówienie.

Zadanie 40 *Jak zmienia się czas wykonania programu klienta w poprzednim przykładzie, w zależności od ustawionego czasu blokowania?*

7.7 Komunikacja między klientem a serwerem

Funkcja clnt_create dla lokalizacji serwera RPC używa numerów 32–bitowych, które nie są związane z przestrzenią numerów portów TCP i UDP, więcej, przy zastosowaniu innych (potencjalnie) protokołów transportowych mogą być mapowane na koordynaty innego rodzaju. Konwersji numerów programu na odpowiednie porty warstwy transportowej dokonuje demon systemowy zwany *portmapperem*, który musi być uruchomiony na stacji serwera. Startujący serwer dokonuje rejestracji udostępnianego programu w bazie portmappera i związania odpowiedniego, dynamicznie określanego portu. Klient, wywołując funkcję clnt_create, kontaktuje się najpierw z portmapperem, który dostarcza mu właściwych danych adresowych serwera. Bazę wiązań portmappera możemy uzyskać dzięki poleceniu `rpcinfo -p nazwa_serwera`.

Komunikacja z samym portmapperem odbywa się również poprzez RPC - z tą różnicą że porty portmappera mają charakter *well-known* - słucha on tradycyjnie na 111-tym porcie TCP i UDP.

7.8 Wady programowania w RPC

RPC jest techniką wygodną i elegancką, pozwalającą nam na podniesienie poziomu abstrakcji w programowaniu rozproszonym ponad specyfikę protokołów sieciowych i ich implementacji oraz usunięcie problemów heterogeniczności. Nie oznacza to jednak, że jest to technika zawsze skuteczna. Wymieńmy główne niedogodności RPC:

- serwery RPC są iteracyjne - w większości implementacji brak wsparcia dla wieloprocesorowości czy wielowątkowości; uzyskuje się je zwykle poprzez trudno przenośne „sztuczki”;
- zwiększenie funkcjonalności serwera RPC odbywa się poprzez mało elegancką manipulację kodem źródłowym wygenerowanym przez prekompilator;
- techniki autoryzujące klienta dla serwera są albo prymitywne, albo w większości implementacji słabo dopracowane;
- w technice nazewniczej brak zabezpieczenia przed zawłaszczeniem numeru;
- ograniczenia interfejsów do wykorzystania jedynie funkcji jednoargumentowych (nie dotyczy to nowszych implementacji RPC) psują przejrzystość programowania;
- pobieranie argumentów i zwracanie rezultatów przez wskaźniki powoduje problemy z zarządzaniem pamięcią;
- kod wynikowy jest tworzony jedynie dla języka C.

Wady te zostały w większości usunięte w projektach bazujących na RPC, a szczególnie w specyfikacji CORBA (*Common Object Request Broker Architecture*) i jej implementacjach.

Rozdział 8

Programowanie w języku Java

8.1 Java a programowanie sieciowe

Java to nie tylko język programowania, ale również środowisko wykonawcze z zestawem bibliotek programistycznych w postaci gotowych do wykorzystania klas. Kod klasy w języku Java jest kompilowany do binarnego pliku, zawierającego ciąg instrukcji wirtualnego procesora, emulowanego przez tzw. *maszynę wirtualną*. Maszyna wirtualna Javy szczelnie ukrywa przed zbiorem swoich aplikacji specyfikę systemu operacyjnego komputera macierzystego, gwarantując przenośność oprogramowania na poziomie binarnym.

Z punktu widzenia programowania sieciowego, środowisko Javy jest bardzo wygodne z następujących przyczyn:

- kod źródłowy oraz wynikowy (binarny) jest niezależny maszynowo - aplikacje mogą swobodnie migrować między heterogenicznymi systemami;
- reprezentacja wewnętrzna danych jest binarnie ujednolicona - programy mogą wymieniać dane bez konwersji;
- standardowe klasy Javy wspierają wielowątkowość;
- synchronizacja w postaci muteksów i zmiennych warunkowych jest elementem języka;
- standardowe klasy Javy implementują interfejs gniazdek;
- standardowe klasy Javy wspierają kod oparty na zdalnym wywołaniu metod (*RMI - Remote Method Invocation*);

- Java „świetnie pasuje” do specyfikacji CORBA.

Analizując powyższe cechy środowiska Javy nie sposób nie zauważyć, że unormowuje ono większość omówionych wcześniej technik charakterystycznych dla środowisk unixowych.

8.2 Realizacja wielowątkowości

Wątek w Javie jest obiektem typu `Thread`. Aby wątek nie wykonywał pustego, domyślnego kodu, należy posłużyć się jedną z dwóch strategii:

- zainicjować obiekt klasy `Thread`, podając w konstruktorze referencję do obiektu klasy implementującej interfejs `Runnable`; jego implementacja jest równoważna z konkretyzacją metody `run`;
- wyprowadzić z klasy `Thread` klasę potomną z przedefiniowaną metodą `run`.

Przykład 50 (Nps1.java) *Program tworzy dwa nowe wątki za pomocą pierwszego sposobu. Obiektami implementującymi interfejs* `Runnable` *są tu instancje klasy głównej. Po utworzeniu dwóch wątków pobocznych funkcja główna rozpoczyna wykonywanie tego samego co one kodu.*

```
public class Nps1 implements Runnable {
  public static void main(String args[]) {
    Thread t1 = new Thread(new Nps1());
    t1.start();
    Thread t2 = new Thread(new Nps1());
    t2.start();
    Nps1 app = new Nps1();
    app.run();
  }
  public void run() {
    for(;;) {
      System.out.print(".");
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {}
    }
```

```
  }
}
```

Wątki (główny i dwa poboczne) wypisują jeden znak kropki na sekundę. Współbieżność powoduje, że na konsoli pojawiają się na sekundę trzy kropki.

Przykład 51 (Nps2.java) *Program poniższy jest analogiczny do tego z poprzedniego przykładu, z tym że wątki tworzone są jako instancje klasy potomnej po* `Thread`. *Dodatkowo przedefiniowano konstruktor, aby przekazać przez niego parametry działania wątku (tu znak do generowania).*

```
public class Nps2 extends Thread {
  char c;
  public static void main(String args[]) {
    Nps2 t1 = new Nps2('1');
    t1.start();
    Nps2 t2 = new Nps2('2');
    t2.start();
    Nps2 app = new Nps2('0');
    app.run();
  }
  public Nps2(char new_c) {
    c = new_c;
  }
  public void run() {
    for(;;) {
      System.out.print(c);
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {}
    }
  }
}
```

Tym razem program generuje na sekundę trzy cyfry: 0 z wątku głównego oraz 1 i 2 z wątków pobocznych. Zwróć uwagę, że kolejność pojawiania się cyfr może się zmieniać.

8.3 Synchronizacja

Synchronizację w Javie przeniesiono na poziom języka. Konstrukcja:

```
synchronized(obiekt) { kod }
```

oznacza, że co najwyżej jeden obiekt może wykonywać analogicznie chroniony kod. Obiekt podany w argumencie sterującym instrukcji `synchronized` ma więc charakter muteksu, choć nie musi on być ani potomkiem jakiegoś specjalnego typu, ani implementować jakiegoś szczególnego interfejsu (wszak wszystkie klasy Javy mają wspólnego przodka, klasę `Object`, z bogatą funkcjonalnością).

Słowo `synchronized` może być używane również jako określenie metody. Metoda tak wyspecyfikowana funkcjonuje tak, jakby w całości była zawarta w bloku `synchronized(this) {}`, co jest równoznaczne z tym, że co najwyżej jeden wątek naraz może aktywować na obiekcie tę metodę, a pozostałe czekają.

Przykład 52 (Nps3.java) *Tablica* `tab` *na pozycji zerowej ma wartość 1, a dalej jest wypełniona zerami. Trzy wątki utworzone przez wątek główny aktywują w pętli na jednym obiekcie metodę zsynchronizowaną* shiftTable. *Metoda przesuwa wartość 1 na jedną pozycję dalej, wstawiając zero na miejsce jej dotychczasowego położenia. Po upływie pięciu sekund wątek główny sygnalizuje poprzez zmianę wartości pola* `stop` *na 1 zatrzymanie wątków dokonujących przesunięć i wyświetla stan tablicy. Program pracuje w pętli nieskończonej.*

```
public class Nps3 implements Runnable {
  public static int[] tab;
  public static int threadcount = 0;
  public static int stop = 0;
  public static void main(String args[]) {
    tab = new int[10];
    tab[0] = 1;
    Nps3 app = new Nps3();
    for(;;) {
      Thread t1 = new Thread(app);
      Thread t2 = new Thread(app);
      Thread t3 = new Thread(app);
      threadcount = 3;
      t1.start();
      t2.start();
```

```
      t3.start();
      try {
        Thread.sleep(5000);
      } catch(InterruptedException e) {}
      stop = 1;
      while(threadcount > 0);
      int i;
      for(i = 0; i < tab.length; i++) {
        System.out.print(tab[i]);
      }
      System.out.println();
      stop = 0;
    }
  }
  public void run() {
    while(stop == 0) {
      shiftTable();
    }
    threadcount--;
  }
  public synchronized void shiftTable() {
    int i;
    for(i = 0; i < tab.length && tab[i] == 0; i++);
    if(i < tab.length) {
      tab[i] = 0;
      tab[(i + 1) % tab.length] = 1;
    }
  }
}
```

Usunięcie `synchronized` *z opisu metody* `shiftTable` *daje niedeterministycznie efekt w postaci powtarzających się jedynek albo całkowicie wyzerowanej tablicy (poczekaj!).*

Zadanie 41 *Który obiekt pełni w tym programie rolę muteksu? Dlaczego wątki konstruuje się z podaniem tego samego obiektu w argumencie, a nie coraz to nowych instancji klasy jak w przykładzie 50?*

Wrażenie zanurzenia w Javie interfejsu wątków posiksowych potęguje zbiór metod klasy `Object`, implementujący zmienne warunkowe (moni-

tory). Metody `wait`, `notify` i `notifyAll` są funkcjonalnymi odpowiednikami funkcji `pthread_cond_wait`, `pthread_cond_signal` i `pthread_cond_broadcast`. Są one bezparametrowe - odpowiednikiem zestawu zmienna warunkowa–muteks jest tu sam obiekt, na rzecz którego je aktywowano (`this`). Konstrukcja *oczekuj na spełnienie warunku* jest najczęściej realizowana przez kod metody:

```
synchronized void wykonajPoSpelnieniuWarunku() {
  while(! warunek) wait();
  ...akcja...
}
```

natomiast zmiana warunku i budzenie oczekujących wątków przez:

```
synchronized void zmienWarunek() {
  ...zmien warunek...
  notifyAll();
}
```

Przykład 53 (Nps4.java) *Wątek główny pozwala na wprowadzanie w pętli liczb całkowitych ze standardowego wejścia. Wątek poboczny konsumuje wprowadzane liczby, nie stosując aktywnego czekania dla synchronizacji z wątkiem głównym.*

```
import java.io.DataInputStream;

public class Nps4 implements Runnable {
  public double x;
  public static void main(String args[]) {
    Nps4 app = new Nps4();
    app.x = 0;
    DataInputStream stdin =
      new DataInputStream(System.in);
    Thread t1 = new Thread(app);
    t1.start();
    for(;;) {
      System.out.print("Wprowadz liczbe: ");
      String s;
      try {
        s = stdin.readLine();
      } catch(java.io.IOException e) {
```

```
        s = "0.0";
      }
      double new_x;
      try {
        new_x = Double.valueOf(s).doubleValue();
      } catch(java.lang.NumberFormatException e) {
        new_x = 0.0;
      }
      app.moznaKonsumowac(new_x);
    }
  }
  public synchronized void run() {
    for(;;) {
      try {
        konsumuj();
      } catch(InterruptedException e) {}
    }
  }
  public synchronized void moznaKonsumowac(double new_x) {
    x = new_x;
    notify();
  }
  public synchronized void konsumuj()
    throws InterruptedException {
    while(x == 0.0) wait();
    System.out.println("Skonsumowano: " + x);
    x = 0;
  }
}
```

Zwróć uwagę na widoczną współbieżność konsumenta i producenta - konsument może wypisywać na konsolę komunikaty, „zamazując” tekst zachęty wątku producenta.

Zadanie 42 *Zsynchronizuj program z poprzedniego przykładu tak, aby producent nie prosił o podanie nowej wartości przed jej konsumpcją przez wątek poboczny. Dla lepszego efektu wprowadź sztuczne opóźnienie w konsumpcji danych.*

8.4 Gniazdka

Interfejs gniazdek został w Javie znacznie uproszczony - wiele czynności, które niemal zawsze wykonuje się w jednakowym porządku zostało zgrupowanych w pojedyncze metody klas *Socket* i *ServerSocket*. Proste pomosty programistyczne, łączące gniazdka ze strumieniami powodują czynią niskopoziomowe programowanie sieciowe czytelnym i łatwym.

Przykład 54 (Nps5.java) *Najprostszy serwer gniazdkowy, zwracający klientowi łańcuch znaków może wyglądać następująco:*

```
import java.io.*;
import java.net.*;

public class Nps5 {
  public static void main(String args[]) {
    try {
      ServerSocket sock = new ServerSocket(10001, 5);
      for(;;) {
        Socket csock = sock.accept();
        DataOutputStream d =
          new DataOutputStream(csock.getOutputStream());
        d.writeChars("Od serwera w Javie...\n");
        csock.close();
      }
    } catch(IOException e) {
      System.out.println("Blad --> " + e);
      System.exit(1);
    }
  }
}
```

Sprawdź serwer łącząc się z nim za pomocą programu `telnet` *lub* `nc`.

Zadanie 43 *Odpowiednio modyfikując program z poprzedniego przykładu (np. zmieniając port na mniejszy od 1024 i wyprowadzając pomocnicze komunikaty na konsolę) ustal, czy funkcja gniazdkowa* `bind` *jest w Javie elementem konstruktora klasy* `ServerSocket` *czy raczej metody* `accept`.

Klient gniazdkowy w Javie jest równie trywialny.

Przykład 55 (Nps6.java) *Program łączy się z serwerem TCP o nazwie podanej w pierwszym argumencie wywołania na porcie z drugiego argumentu. Czyta oferowane mu dane i wypisuje na standardowe wyjście.*

```
import java.io.*;
import java.net.*;

public class Nps6 {
  public static void main(String args[]) {
    if(args.length != 2) {
      System.out.println("Uruchom: java Nps6 serwer port");
      System.exit(1);
    }
    try {
      InetAddress addr = InetAddress.getByName(args[0]);
      Socket sock = new Socket(addr.getHostName(),
                       Integer.valueOf(args[1]).intValue(),
                       true);
      DataInputStream d =
        new DataInputStream(sock.getInputStream());
      String s = d.readLine();
      sock.close();
      System.out.println("Przeczytalem: " + s);
    } catch(Exception e) {
      System.out.println("Blad --> " + e);
      System.exit(1);
    }
  }
}
```

Sprawdź jego działanie z serwerem z poprzedniego przykładu.

Zadanie 44 *Napisz w Javie przykładowy gniazdkowy serwer współbieżny oraz testującego go współbieżnego klienta (o funkcjonalności programu z przykładu 32).*

8.5 RMI

Technologia RMI jest w języku Java odpowiednikiem RPC. Cechy języka i środowiska wykonawczego Javy powodują, że technologia ta jest dużo bardziej dojrzała

i elegancka. Z punktu widzenia programisty podstawowe różnice w obu technikach można wymienić w następujących punktach:

- w RMI (podobnie jak w CORBIE) mamy do czynienia ze zdalnym obiektem (choć różnica w stosunku do RPC jest dość ulotna);
- ponieważ Java standaryzuje binarną reprezentację typów nie jest wymagana warstwa kodująca argumenty i rezultaty na postać sieciową (XDR w RPC);
- z tego samego powodu nie jest wymagane uogólnianie interfejsu w języku, który jest dopiero tłumaczony na właściwy kod na danej platformie - interfejsy w RMI są po prostu pisane w Javie;
- nie pojawiają się żadne sztuczne ograniczenia na liczbę argumentów i ich typ;
- implementowanie serwera i klienta odbywa się w drodze dziedziczenia (identycznie jest w specyfikacji CORBA), a nie dopisywania ciała do wygenerowanych prototypów;
- usługi lokalizujące obiekt są dużo bardziej wyrafinowane niż portmapper z RPC;
- dyspozytorem kodu jest pośrednik (ang. *broker*) obiektów, a nie wygenerowany pieniek serwera; nie ma żadnej konieczności modyfikacji pieńka serwera w celu uzyskania jakiejś szczególnej funkcjonalności;
- istnieje możliwość precyzyjnego określenia praw dostępu do zdalnego obiektu.

Proces tworzenia oprogramowania z wykorzystaniem RMI przedstawia poniższy przykład.

Przykład 56 (Nps7.java/Nps7Impl.java/Nps7.policy/Nps7Clnt.java) *Interfejs serwera zdalnego obiektu wygląda następująco:*

```
import java.rmi.*;
public interface Nps7 extends Remote {
  double dodaj(double x, double y)
    throws RemoteException;
}
```

Obiekt zdalny (potomek klasy `Remote`*) udostępnia tu jedną metodę, służącą do dodawania dwóch liczb zmiennoprzecinkowych. Jego implementacja może wyglądać tak:*

```
import java.rmi.*;
import java.rmi.server.*;

public class Nps7Impl extends UnicastRemoteObject
  implements Nps7 {
  public Nps7Impl() throws RemoteException {
    super();
  }
  public double dodaj(double x, double y) {
    System.out.println("Realizuje zamowione dodawanie\n");
    return x + y;
  }
  public static void main(String args[]) {
    if(System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
    try {
      Nps7Impl obj = new Nps7Impl();
      Naming.rebind("//localhost/Nps", obj);
    } catch(Exception e) {
      System.out.println("Blad --> " + e);
    }
  }
}
```

Mamy tu nowy konstruktor, zachowujący funkcjonalność odpowiedniej metody przodka (jego definicja jest niezbędna ze względu na potencjalnie generowany wyjątek `RemoteException`*). Jest również implementacja samego dodawania (wraz z diagnostycznym komunikatem). W końcu funkcja główna, ustalająca tzw.* zarządcę bezpieczeństwa, *tworząca instancję klasy i dowiązująca ją do odpowiedniej nazwy w formacie UNC. Nazwa ta wskazuje na serwer lokalny, co oznacza że klasa implementacji oraz broker będą pracowały na tej samej maszynie.*

Po kompilacji obu plików, poleceniem `rmic Nps7Impl`*, wytwarzamy dwa nowe pliki klas: tzw.* szkielet *(ang.* skeleton, *tu w pliku* `Nps7Impl_Skel.class`*) oraz* pieniek *(ang.* stub, `Nps7Impl_Stub.class`*). Ten drugi powinniśmy dostarczyć programiście klienta (w tym sensie jest on odpowiednikiem pliku* `.x` *z RPC, choć nie wymaga prekompilacji).*

Broker RMI (o ile nie był uruchomiony wcześniej) uruchamiamy poleceniem `rmiregistry` *(mogą nasuwać się skojarzenia z portmapperem w RPC, aczkolwiek zadaniem tego ostatniego jest jedynie świadczenie usług nazewniczych). Broker jest serwer, więc wykonać to należy na osobnej konsoli (będziemy na niej obserwowali komunikaty diagnostyczne). Rejestracji zdalnego obiektu dokonujemy w brokerze za pomocą uruchomienia implementacji zdalnego obiektu:*

```
java -Djava.security.policy=Nps7.policy Nps7Impl
```

Własność `java.security.policy` *została w trakcie wywołania ustawiona na plik* `Nps7.policy`*, zawierający zbiór praw do zdalnego obiektu (tu: wszystkie prawa dozwolone):*

```
grant {
  permission java.security.AllPermission;
};
```

Klient zdalnego obiektu może mieć kod następujący:

```
import java.rmi.*;

public class Nps7Clnt {
  public static void main(String args[]) {
    Nps7 ro = null;
    try {
      ro = (Nps7) Naming.Lookup("//localhost/Nps");
      System.out.println("1.1 + 2.3 = "
                         + ro.dodaj(1.1, 2.3));
    } catch(Exception e) {
      System.out.println("Blad --> " + e);
    }
  }
}
```

Zadanie 45 *Rozbuduj obiekt zdalny* `Nps7` *o kolejną funkcję arytmetyczną.*

Literatura

Podstawową pozycją bibliograficzną jest oczywiście podręcznik W. R. Stevensa „UNIX – programowanie usług sieciowych". Z racji obszerności tego podręcznika (dwa potężne tomy!) zamieszczam tu referencje rozdziałów mojego kursu do odpowiednich rozdziałów wydania WNT z roku 1999.

- treść rozdziału 3 jest rozproszona po całym podręczniku, głównie natomiast w rozdziale 1 tomu 2; opis interfejsu sygnałów oraz zagadnień związanych z kończeniem procesu potomnego znajdziemy w rozdziale 5 tomu 1 (5.8 – 5.10);
- zagadnienia poruszane w rozdziale 4 są rozsiane po tomie 2: sekcja 4.1 – sekcja 4.3, sekcja 4.2 – sekcja 4.6, sekcja 4.3 – rozdział 3, sekcja 4.4 – rozdział 6, sekcja 4.5 – rozdział 14, sekcja 4.6 – rozdział 12, sekcja 4.7 – rozdział 11;
- rozwinięcie treści rozdziału 5 można znaleźć w tomie 1 rozdział 23;
- wiadomości przedstawione w rozdziale 6 to właściwie materiał całego tomu 1 podręcznika; najwięcej informacji znajdzie czytelnik w sekcjach 3.2 (sekcja 6.2 kursu), 4.2 (6.3), 4.3 (6.4), 4.4 – 4.6 (6.5), 8.2 (6.6), 4.7 – 4.8 (6.7), 12.5 (6.8), 6.3 (6.11) oraz w rozdziałach 9 (6.9) i 14 (6.10);
- treści rozdziału 7 znajdziemy w tomie 2 w rozdziale 16.

Alternatywną pozycją jest tu książka M. Gabassiego i B. Dupouy „Przetwarzanie rozproszone w systemie UNIX" (Lupus 1996).

Programowanie w środowisku systemu Unix dobrze opisuje podręcznik autorstwa W. R. Stevensa: „Advanced Programming in the UNIX Environment" (Addison–Wesley 1992).

Wiedzę zawartą w rozdziale 8 czytelnik może poszerzyć korzystając z książki B. Boone'a „Java dla programistów C i C++" (WNT 1998). Sekcja 8.5 może być zgłębiona za pomocą dokumentacji elektronicznej, dostarczonej z kursem.

Spis treści