Na prawach rękopisu

KATEDRA INFORMATYKI TECHNICZNEJ
POLITECHNIKA WROCŁAWSKA

# Programowanie Współbieżne

Tomasz Surmacz

Słowa kluczowe:

UNIX
programowanie
współbieżność
semafory
pamięć współdzielona
komunikacja międzyzadaniowa
gcc
vi, vim
makefile

Wrocław, 2 marca 2021

# 1 Tematyka zajęć

Nabycie praktyki dotyczącej tematów poruszanych na wykładzie. W szczególności:

- Podstawowa obsługa systemu UNIX (Linux)
- Kompilowanie i uruchamianie programów w systemie UNIX (edytor `vi`/`vim`, program `make` i pliki *Makefile*, kompilator `gcc`)
- Narzędzia wspomagające i systemy kontroli wersji - `git` oraz `GitHub`
- Wejście i wyjście w systemie UNIX, procesy, kontekst procesu, sterowanie procesami, wątki i programy wielowątkowe
- Strumienie pipe i FIFO
- Mechanizmy IPC System V: semafory, pamięć wspólna
- Aplikacje wielowątkowe i mechanizmy synchronizacji: semafory, monitory, zmienne warunkowe, problemy współbieżności
- Sygnały

Programy pisane w trakcie zajęć powinny być pisane w języku C i kompilować się za pomocą `gcc -ansi` lub `gcc -std=c99`.

# 2 Edytor `vi`/`vim`

VI/VIM to bardzo dobry edytor, nadający się szczególnie do edycji programów, choć nie tylko. Jest to także podstawowy edytor w każdym systemie UNIX (nie licząc jego poprzednika – edytora `ed`), można mieć więc pewność, że będzie on dostępny zawsze, bez względu na to, jakiego systemu UNIX aktualnie używamy[1].

Sposób edycji tekstów może wydawać się na początku nieco dziwny[2], łatwo się jednak do niego przyzwyczaić. Podstawowa cecha to fakt, że istnieją dwa podstawowe tryby pracy – tryb *komend* i tryb *edycji*. W trybie

[1] W każdym systemie na pewno dostępny będzie edytor `vi`, z którego wywodzi się cała rodzina kompatybilnych implementacji. Edytor `vim` powstał najpierw jako *VI iMitation* na komputery Amiga, ale szybko przewyższył funkcjonalnością oryginalny edytor `vi`, dlatego obecnie jego nazwa tłumaczona jest na *VI iMproved*. Mimo to, w dalszym ciągu jest on w 99% zgodny z oryginalnym `vi` (po ustawieniu opcji `:set compatible`. Więcej na temat różnic pomiędzy obu edytorami można znaleźć pisząc w `vim`-ie `:help vi-differences`

[2] szczególnie w porównaniu z typowymi edytorami działającymi w systemach DOS/Windows.

komend prawie każdy klawisz (także litery i cyfry) zamiast powodować wpisywanie tekstu, powoduje wykonanie odpowiadającej mu komendy. Dopiero przejście do trybu edycji pozwala na wpisywanie tekstu, uniemożliwiając jednocześnie wykonywanie większości komend, aż do czasu ponownego przejścia do trybu komend, co zawsze następuje po naciśnięciu klawisza ESC[3]. Jest to efektem innej cechy edytora vi – wszystkie komendy możliwe są do wykonania za pomocą standardowych klawiszy, dostępnych na *każdej* klawiaturze dowolnego terminala. Bez względu na to, czy z systemu UNIX korzystamy na jego konsoli, czy logując się przez sieć z lepszych lub gorszych emulatorów terminali, czy w końcu ze sprzętowych terminali takich jak terminale VT100 czy FACIT, zawsze będzie możliwe dokonanie edycji pliku za pomocą vi, mimo braku klawiszy funkcyjnych, czy nawet klawiszy strzałek.

Istnieje wiele wersji edytora vi – od „surowego" vi dostępnego w każdym systemie UNIX, nvi domyślnie instalowanego w systemach BSD, po vim (vi-enhanced) i gvim (GUI vim). Praca za pomocą vim jest zdecydowanie wygodniejsza, a edytor ten dostępny jest w postaci kodu źródłowego. Jest też instalowany domyślnie we wszystkich systemach typu Linux, istnieją też implementacje na wszystkie wersje systemów UNIX, a także MS DOS, MS-Windows, AmigaOS, Atari MiNT, OS/2, OS/390, MacOS, OS/X, OpenVMS, RISC OS i QNX. Edytor vim uruchomiony pod nazwą vi zachowuje kompatybilność ze „starym" vi, dlatego aby wykorzystać wszystkie jego udogodnienia, należy go uruchamiać pisząc pełną nazwę – vim. Dalsza część instrukcji dotyczy zarówno edytora vi jak i vim.

## 2.1 Samouczek vi

Poznanie edytora vi warto zacząć od specjalnego samouczka. Najlepiej po prostu napisać z linii komend vimtutor, co spowoduje uruchomienie edytora z wczytanym tekstem samouczka – jest to zwykły plik tekstowy zawierający instrukcje do wykonania, który można czytać i edytować, eksperymentując z programem vi i vim. Można też sobie ten samouczek zapisać we własnym katalogu domowym, a następnie otwierać go za pomocą vi, podając jako argument nazwę pliku, np.:

```
cd
vi vi-tutor.txt                    (lub:  vim vi-tutor)
```

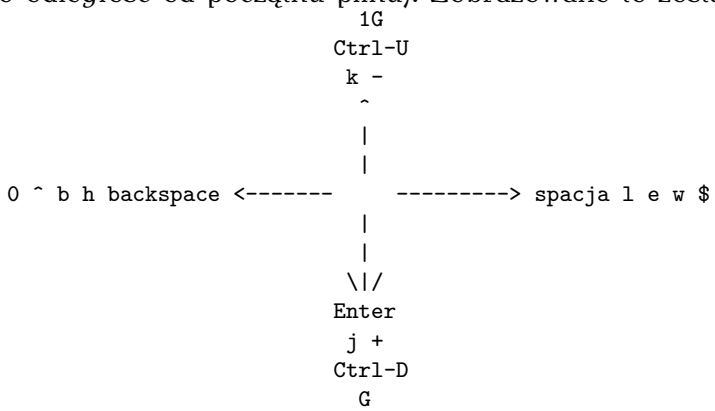
Plik ten należy czytać, wykonując jednocześnie zawarte w nim instrukcje, ćwicząc w ten sposób edytowanie tekstów za pomocą edytora vi.

[3] warto przy okazji zapamiętać, że naciśnięcie ESC jest zawsze bezpieczną czynnością w vi, tzn. w trybie edycji powoduje przejście do trybu komend, natomiast podczas wpisywania komendy złożonej przerywa ją i wraca do trybu komend, a w trybie komend – wysyła do terminala krótki pisk, sygnalizujący, że vi już był w trybie komend.

Jedna z pierwszych lekcji omawia sposób wychodzenia z edytora (co na początku nie jest rzeczą trywialną), następnie pojawiają się komendy służące do poruszania się po tekście, a potem coraz bardziej zaawansowane polecenia edycji tekstu.

## 2.2 Podstawowe komendy edytora vi

W celu poruszania się po tekście (w trybie komend oczywiście) można użyć jednego z czterech klawiszy: h, j, k i l. Powodują one odpowiednio przesuwanie kursora o jeden znak lub linię w lewo, dół, górę i prawo. W tym samym celu można także użyć alternatywnych klawiszy -, +, Backspace, Spacja i Enter. Można także użyć klawiszy strzałek, jednak na niektórych typach terminali strzałki mogą generować kody, które nie zostaną rozpoznane po stronie systemu UNIX, dlatego warto znać komendy wymienione przed chwilą. W celu znalezienia się na początku linii można nacisnąć klawisze 0 lub ^, a na końcu linii – znak $. Przesuwać się po pół ekranu w górę lub w dół można naciskając odpowiednio Ctrl-U lub Ctrl-D, natomiast poruszać się po tekście od słowa do słowa można za pomocą klawiszy e, w i b (a także E, W i B – subtelna różnica w działaniu, w traktowaniu co jest słowem). Na początek lub koniec tekstu można się przenieść naciskając 1G lub G (dokładniej: komenda *n*G ustawia kursor w linii numer *n*, analogicznie – komenda *n*% przeniesie kursor do miejsca wyznaczonego procentowo jako odległość od początku pliku). Zobrazowane to zostało poniżej:




W celu wejścia do trybu edycji można użyć jednego z następujących klawiszy:

- i rozpoczęcie wpisywania tekstu od miejsca, w którym znajduje się kursor.
- I rozpoczęcie wpisywania tekstu przed pierwszym znakiem w linii,

w której znajduje się kursor. To samo można uzyskać naciskając ^ i.

a rozpoczęcie wpisywania tekstu za znakiem, za którym znajduje się kursor.

A rozpoczęcie wpisywania tekstu za ostatnim znakiem w linii, w której znajduje się kursor. To samo można uzyskać naciskając $ a.

o utworzenie nowej, pustej linii, za linią, w której znajduje się kursor, i rozpoczęcie wpisywania tekstu na jej początku. Równoznaczne z $ a Enter.

O utworzenie nowej, pustej linii, przed linią, w której znajduje się kursor, i rozpoczęcie wpisywania tekstu na jej początku.

S zastąpienie tekstu od aktualnej pozycji kursora do końca linii tym, co zostanie zaraz wpisane (po usunięciu tekstu lub zaznaczeniu znakiem $ tego, co zostanie skasowane, edytor automatycznie przechodzi do trybu wpisywania tekstu).

R wejście do trybu edycji jak komendą i, ale w trybie nadpisywania znaków. W edytorze vim w trybie edycji pomiędzy trybem nadpisywania i dopisywania tekstu można przechodzić naciskając klawisz Ins.

## 2.3 Komendy pomocne w edycji programów w C

vi zapewnia wiele narzędzi pomocnych przy edycji programów w C i innych językach programowania. W celu znalezienia nawiasu pasującego do pary w konstrukcjach takich, jak:

```
if ((f[idx]=fopen("plik", "rw"))==NULL) {printf("błąd\n")};
```

wystarczy ustawić kursor na odpowiednim nawiasie, po czym nacisnąć znak %. Jeżeli natomiast zostanie ustawiona zmienna showmatch (za pomocą komendy :set showmatch lub :set sm), już w trakcie wpisywania tekstu, w momencie zamykania dowolnego nawiasu, kursor na chwilę przeskoczy do odpowiadającego mu nawiasu otwierającego, pozwalając szybko zorientować się, który poziom zagnieżdżenia właśnie jest zamykany. Funkcję tę wyłączyć można za pomocą :set noshowmatch lub :set nosm.

Często wykonywaną operacją jest także przesuwanie bloku programu w lewo lub w prawo – dodając lub likwidując wcięcie, aby odzwierciedlić strukturę programu. W tym celu posłużyć się można komendami `<` i `>`. Są to komendy z grupy wymagającej podania argumentu określającego zakres ich działania, tak więc przykładowo `>` `>` spowoduje dodanie wcięcia w aktualnie edytowanej linii, a `>` `'` `a` we wszystkich liniach od aktualnie edytowanej do linii, w której znajduje się znacznik „a", ustawiony tam wcześniej komendą `m` `a` (znaczników takich można ustawić 26 – tyle, ile jest liter alfabetu łacińskiego). Jeżeli kursor znajduje się na nawiasie otwierającym lub zamykającym pewien blok, np. w konstrukcji `if`, `for`, `while`, itp., cały blok można przesunąć łącząc komendę `>` lub `<` z komendą `%`, ustawiając kursor na odpowiednim nawiasie i wpisując np. `<` `%`.

Innymi często używanymi komendami są `d` `w` lub `c` `w`, czyli „delete word" lub „change word" – powodują odpowiednio skasowanie tekstu od kursora do początku następnego słowa lub skasowanie i od razu wejście do trybu edycji, czyli zamiana tego słowa na inne.

Automatyczne uzupełnianie wcięć po naciśnięciu klawisza `Enter` (do takiego poziomu, jak w poprzedniej linii) można uzyskać wpisując `:set autoindent` lub `:set ai`. Podczas wpisywania tekstu, po przejściu do nowej linii, pojedyncze wcięcie można zlikwidować naciskając `^D` (nie wychodząc z trybu wpisywania tekstu), a na stałe wyłączyć wpisując `:set noautoindent` lub `:set noai` czy `:set nosi`. W edytorze `vim` działają także opcje `:set smartindent` oraz `:set cindent`, które w jeszcze lepszy sposób potrafią dopasowywać wcięcia, dopasowując je do analizowanego na bieżąco tekstu (zgodnie ze składnią typowych języków programowania (smart) lub konkretnie języka C).

Do szybkiego poruszania się po treści programu można też użyć komend `[` `[` i `]` `]`, które powodują przemieszczanie kursora na początek poprzedniej lub następnej funkcji.

Bardzo przydatną funkcją występującą w nowszych wersjach edytora `vim` jest także możliwość podświetlania składni (zależnie od typu terminala stosowane są kolory lub tylko zmiana intensywności wyświetlanego tekstu). Funkcję tę można włączyć pisząc `:syntax on`. Jeśli z jakiegoś powodu `vim` nie rozpozna wg jakich reguł składniowych powinien podświetlać edytowny tekst (np. używamy nietypowego rozszerzenia pliku), to za pomocą `:set syntax=c` możemy wprost zdefiniować jakiej składni odpowiada edytowany tekst. Komendę taką można umieścić nawet bezpośrednio w edytowanym pliku, na jego początku, jak to pokazano w rozdziale 2.5.

## 2.4 Zmienne edytora vi

Oprócz omówionych w poprzednim rozdziale zmiennych, często przydatne są następujące zmienne edytora vi:

- ts=*wartość* lub tabstop=*wartość*

  Określa ile spacji należy drukować zamiast pojedynczego tabulatora. Standardowo jest to 8, jednak w celu poprawienia czytelności programu (aby długie linie zbyt szybko „nie wychodziły za ekran") można tę wartość zmniejszyć np. na 4 lub 2. Analogicznie do wartości ts należy ustawić wartość zmiennej sw (shiftwidth), która odpowiada za to, o ile spacji cofnie się kursor, gdy będziemy kasować jakieś wcięcie.

- wm=*wartość* lub wrapmargin=*wartość*

  Określa na ile kolumn przed prawym marginesem ekranu/okna vi ma automatycznie generować znak przejścia do nowej linii. Standardowo jest to 8 (co przy 80-kolumnowych konsolach oznacza pisanie w wierszach po ok. 72 znaki). Ustawienie tej zmiennej na wartość 0 wyłącza automatyczne przenoszenie tekstu do nowej linii – przydatne właśnie w pisaniu wszelkiego rodzaju programów.

- nu lub number

  Włącza pokazywanie przez edytor numeru linii. Cały tekst na ekranie zostaje przesunięty o kilka kolumn w prawo, a każda linia zostaje poprzedzona jej numerem. Numery te są jedynie wyświetlane na ekranie i nie są umieszczane w pliku po jego zachowaniu na dysk. Opcję tę można wyłączyć wpisując :set nonu lub :set nonumber.

- sm lub showmatch

  Po ustawieniu tej opcji, podczas wpisywania tekstu w trybie edycji, po wpisaniu dowolnego nawiasu zamykającego, kursor na chwilę zostanie przeniesiony do pasującego nawiasu otwierającego, pozwalając zorientować się, który nawias właśnie został zamknięty. Jeśli nie można znaleźć nawiasu otwierającego, generowany jest ostrzegawczy dźwięk. Opcję można wyłączyć wpisując :set nosm lub :set noshowmatch.

- showmode

  Ustawienie tej opcji powoduje, że vi w ostatniej linii ekranu pokazuje, w jakim trybie się znajduje. Jeśli jest to tryb edycji, pojawi się tam tekst „-- INSERT --" lub „-- REPLACE --. Po wyjściu do trybu komend wyświetlany tekst znika. Opcję można wyłączyć wpisując :set noshowmode.

## 2.5 Pliki konfiguracyjne *.exrc* i *.vimrc*

Definicje zmiennych, (komendy `:set`), definiowanie mapowania klawiszy za pomocą komend `:map` i `:map!` i inne komendy można umieścić w pliku *.exrc*. Jeżeli program `vi` podczas uruchamiania znajdzie w katalogu bieżącym lub w katalogu domowym użytkownika taki plik, to zinterpretuje jego zawartość jako komendy do wykonania – tym samym ustawiając odpowiednie zmienne, definiując środowisko pracy, do którego przyzwyczajony jest użytkownik.

W przypadku edytora `vim` w pierwszym rzędzie czytany jest plik *.vimrc*, a dopiero wtedy, gdy taki nie zostanie znaleziony – „standardowy" *.exrc*. Oprócz tego, komendy `:set` można także umieszczać bezpośrednio w edytowanych plikach. Jeśli chcemy tak zrobić, to w pierwszych lub ostatnich 512 bajtach pliku musi się znaleźć tekst postaci:

```
vim:set ....komendy... :
```

Czyli tekst `vim:set`, a następnie ustawienia poszczególnych zmiennych, zakończone dwukropkiem. Wszystko to musi się znaleźć w jednej linii tekstu. Przykładowo, w programach pisanych w języku C przydatne ustawienia wpisane bezpośrednio w pliku mogą wyglądać tak:

```
/* vim:set noai nu sm wm=0 ts=4 sw=4 nowrap syntax=c:
 */
```

(aby taki wpis był skuteczny, „vim" musi być osobnym słowem, poprzedzonym spacją. Podobnie po końcowym dwukropku powinien występować znak końca linii lub co najmniej jedna spacja i dopiero kolejne znaki).

# 3 Kompilowanie programów za pomocą `gcc`

Do kompilacji programów pisanych w językach C i C++ służy kompilator `gcc`. Dostępny jest on jednak pod kilkoma nazwami – także jako `g++` – i zależnie od użytej nazwy potrafi inaczej traktować kompilowane pliki, pozwalając lub nie, na użycie rozszerzeń języka C++ w programach pisanych w C itp. To, czy program jest pisany w C, czy C++ jest w pierwszym rzędzie rozpoznawane przez rozszerzenie pliku. Programy pisane w C powinny mieć rozszerzenie *.c*, podczas gdy programy pisane w C++ powinny mieć rozszerzenie *.C*, *.cxx* lub *.c++*. Ponadto wywołanie `gcc` poprzez użycie nazwy `g++` zawsze powoduje kompilację programu w trybie C++. Kompilację ściśle wg reguł języka C (ignorując

np. iostreams dostępne w C++) można osiągnąć korzystając z opcji gcc -ansi

## 3.1 Opcje kompilacji

W najprostszym przypadku wywołanie gcc lub g++ wymaga jednego argumentu, którym jest nazwa kompilowanego pliku źródłowego, np.:

```
gcc helloworld.c
```

W wyniku powyższej komendy gcc skompiluje program *helloworld.c* do tymczasowego pliku pośredniego, który następnie zlinkuje, tworząc program wynikowy w pliku *a.out*[4]. Tak będzie zawsze, gdy nazwa pliku wykonywalnego nie zostanie wyraźnie podana jako argument podczas wywoływania gcc. Aby to zmienić, wystarczy dołączyć opcję -o, podając także nazwę pliku, np.:

```
gcc -o helloworld helloworld.c
```

Nazwa pliku wykonywalnego zwykle odpowiada nazwie pliku źródłowego, ale nie musi tak wcale być, szczególnie w przypadkach, gdy źródła programu znajdują się w kilku plikach. W powyższym przykładzie gcc dokonuje równocześnie kompilacji i konsolidacji (linkowania) programu, czynności te można jednak rozdzielić. Dodanie opcji -c powoduje, że plik źródłowy zostaje skompilowany[5], a plik pośredni (plik „obiektowy") otrzymuje taką samą nazwę, jak oryginalny, lecz z rozszerzeniem .o. Aby otrzymać plik wykonywalny, należy użyć programu ld lub ponownie gcc,który będzie tym razem działał w trybie konsolidacji, wywołując w rzeczywistości program ld. W takim trybie gcc zadziała automatycznie, jeśli lista plików do „kompilacji" będzie się składała z plików z rozszerzeniem .o.

```
gcc -o helloworld.o -c helloworld.c
gcc -o helloworld helloworld.o
```

lub:

[4] Standardowo system UNIX przy uruchamianiu programów nie przeszukuje w tym celu bieżącego katalogu. Aby uruchomić program znajdujący się w katalogu bieżącym należy albo dodać katalog . (pojedyncza kropka) na końcu swojej ścieżki, czyli zmiennej path, albo uruchomić program podając pełną ścieżkę do niego, która może rozpoczynać się od katalogu bieżącego: ./a.out.

[5] tak naprawdę, najpierw zostaje uruchomiony preprocesor cpp, który usuwa z plików źródłowych komentarze i interpretuje wszystkie makra zaczynające się znakiem #, takie jak #define, #include, #if, itp., lecz z naszego punktu widzenia jest to na razie nieistotne.

```
gcc -c helloworld.c
gcc -o helloworld helloworld.o
```

Podczas kompilacji przydatna może być także opcja `-I`, mówiąca gdzie należy szukać plików z rozszerzeniem *.h* włączanych przy pomocy `#include <plik.h>` lub `#include "plik"`. Przy pomocy opcji `-D` można zdefiniować dodatkowe symbole, których istnienie lub brak może powodować warunkową kompilację fragmentów programu (poprzez użycie odpowiednich `#ifdef SYMBOL` lub `#if defined(SYMBOL)`.

Bardzo przydatne może być także użycie następujących opcji:

```
gcc -Wall -g -o helloworld -c helloworld.c
```

Opcja `-Wall` włącza drukowanie przez kompilator wszystkich ostrzeżeń, takich jak deklarowanie funkcji przez jej użycie (czyli brak właściwego pliku nagłówkowego i prototypu funkcji), użycie niewłaściwej liczby argumentów funkcji lub użycie argumentów niewłaściwych typów. Opcja `-g` powoduje, że do pliku wykonywalnego zostaną dołączone informacje o nazwach symboli użytych w programie (nazwach zmiennych, funkcji, numerach linii, itd.), pomocne w debugowaniu programu.

Podczas konsolidacji programu przydatne są natomiast opcje mówiące jakie dodatkowe biblioteki są potrzebne do prawidłowego wygenerowania pliku wynikowego (opcja `-l`) oraz gdzie takich bibliotek szukać (opcja `-L`). Przykładowe użycie tych opcji zostało przedstawione poniżej:

```
gcc -g -c -DUNIX -DVER="5" -I.. -I/usr/local/X11/include helloagain.c
gcc -g -o helloagain -L/usr/local/X11/lib helloagain.o -lX11 -lm -lsocket
```

Pierwsza instrukcja powoduje zdefiniowanie symboli „UNIX" i „VER" przed wywołaniem preprocesora/kompilatora (jest to równoważne umieszczeniu linii `#define UNIX` i `#define VER 5` na początku kompilowanego programu) oraz dołączenie katalogów *..* i */usr/local/X11/include* (w tej kolejności) do listy katalogów, w których znajdują się pliki *.h*. Standardowo, lista ta zawiera katalogu */usr/include* i */usr/local/include*.

Druga instrukcja uruchamia `gcc` w trybie konsolidacji, każąc„zlinkować" plik *helloagain.o*, tworząc plik wynikowy *helloagain*, a podczas tego procesu użyte zostają biblioteki `libX11` (funkcje obsługi systemu X-window), `libm` (biblioteka zawierające funkcje matematyczne) i `libsocket` (funkcje komunikacji sieciowej za pomocą gniazdek[6]). Mimo, iż podawana jest

[6]biblioteka ta konieczna jest w systemach V (np. w Solarisie zainstalowanym na serwerze cyber). W systemach BSD lub z podsystemem sieciowym opartym na BSD (takich

tylko „krótka" nazwa biblioteki, to w celu znalezienia rzeczywistego pliku zawierającego funkcje tej biblioteki, do jej nazwy na początku doklejany jest człon *lib*, a na końcu – rozszerzenie *.a* (linkowanie statyczne) lub *.so* (linkowanie dynamiczne). Zwykle także nie jest konieczne podawanie opcji `-L`, gdyż standardowe biblioteki znajdują się w jednym z katalogów */lib* lub */usr/lib*, które są zawsze przeszukiwane przez linker (zarówno `gcc` jak i `ld`). Użycie opcji `-lm` powyżej oznacza więc konieczność dołączenia biblioteki znajdującej się w pliku *libm.so* lub *libm.a*, który można znaleźć w jednym z katalogów */lib*, */usr/lib* lub */usr/local/X11/lib*.

## 3.2 Typowe błędy i ostrzeżenia

- `implicit declaration of function 'foo()'`

  Deklaracja funkcji następuje przez jej użycie, a więc nie został zdefiniowany prototyp funkcji, a co za tym idzie – nie będzie możliwa kontrola typów argumentów przez kompilator. Jeśli ostrzeżenie dotyczy funkcji systemowej, oznacza to, że zabrakło odpowiedniej dyrektywy `#include`. Nazwa odpowiedniego pliku systemowego jest zawsze wymieniona na stronie dokumentacji systemowej, w podanym przykładzie należałoby więc sprawdzić `man foo`, a w razie znalezienia komendy o podanej nazwie w rozdziale pierwszym, przejrzenie także rozdziałów 2 lub 3 (np. `man 2 foo`).

- `control reaches end of non-void function`

  Ostrzeżenie, mówiące że definiowana funkcja nie została zadeklarowana z użyciem slowa kluczowego „`void`" (lub w ogóle nie został zadeklarowany żaden typ zwracanej wartości, wobec czego kompilator przyjął typ „`int`"), więc powinna zwracać jakąś wartość, tymczasem brak w niej wywołania *return()* z odpowiednią wartością, lub poprzez rozgałęzienia w konstrukcjach `if (...) ... else ...` możliwe jest zakończenie funkcji bez ustalenia zwracanej wartości. Jeśli funkcja nie zwraca żadnej wartości, należy ją zadeklarować jako `void foo(...);`, jeśli zwraca – dopisać brakujące wywołania *return()*.

- `passing arg 1 of 'foo' makes pointer from integer without a cast`

  Podczas przekazywania parametrów do funkcji (w tym przypadku jest to pierwszy argument funkcji) następuje niejawna zmiana typu – zamiast wskaźnika na zmienną typu `int`, przekazywana jest wartość zmiennej. Należy przyjrzeć się definicji funkcji i przekazać dane takiego typu, jakiego wymaga wywołanie funkcji.

jak Linux) funkcje obsługujące komunikację sieciową umieszczone zostały w bibliotece `libc`, dołączanej w sposób niejawny do każdego programu.

# 4 Pliki *Makefile*

W celu ułatwienia sobie kompilacji programów można skorzystać z programu `make` i pliku *Makefile* (lub *makefile*, który jest czytany po wydaniu polecenia `make`. Plik taki zawiera instrukcje mówiące co program `make` ma zbudować w bieżącym katalogu i w jaki sposób. Zwykle będą to reguły budowania programu wynikowego poprzez kompilację plików źródłowych pisanych w języku C, Pascal lub innym, mogą to jednak być instrukcje powodujące wykonanie dowolnych innych programów, niekoniecznie kompilatorów.

W najprostszym pliku *Makefile* wystarczy wymienić dwie rzeczy – cel, czyli nazwę pliku wynikowego, oraz zależności – czyli nazwę pliku lub plików, które są konieczne do zbudowania pliku wynikowego. Jeżeli źródła programu znajdują się w pliku *program1.c*, wystarczy wpisać:

```
program1: program1.o
```

Reguła ta oznacza, że do zbudowania pliku *program1* potrzebny jest plik *program1.o*, natomiast nie są podane szczegółowe instrukcje jak mając do dyspozycji plik z rozszerzeniem *.o* zlinkować go, tworząc program wykonywalny. Nie został także przedstawiony sposób utworzenia pliku *program1.o* poprzez kompilację pliku *program1.c*. Nie jest to konieczne, gdyż program `make` w celu kompilacji skorzysta także z niejawnie definiowanych zależności opisanych poniżej:

```
%.o: %.c
        $(CC) $(CFLAGS) -o $@ $*

%: %.o
        $(CC) $(LD_FLAGS) -o $@ $*
```

`$(CC)` jest odwołaniem się do wartości zmiennej `CC`, która standardowo zawiera tekst `cc` lub `gcc`, czyli powoduje wywołanie kompilatora języka C. Zmienne `CFLAGS` i `LD_FLAGS` z definicji są puste, można jednak nadać im wartości definiując je wcześniej, tym samym przekazując odpowiednie opcje podczas kompilacji lub linkowania programów. Symbole `$@` i `$*` zostają odpowiednio zamienione na to, co wystąpiło po lewej i po prawej stronie znaku ':' w definiowanej zależności, w miejscu odwołań zawierających znaki '%'.

Program `make` przed próbą zbudowania pliku wynikowego sprawdza datę modyfikacji tego pliku na dysku, a także daty modyfikacji wszystkich plików znajdujących się po stronie zależności i jeśli plik wynikowy jest nowszy niż każdy z plików źródłowych, uznaje, że jest to aktualna wersja

i nic nie trzeba robić. Plik wynikowy zostanie więc przebudowany tylko wtedy, gdy co najmniej jeden z plików potrzebnych do jego zbudowania jest młodszy niż on sam (lub nie istnieje, bo to oznacza, że trzeba go zbudować, a wówczas data modyfikacji będzie na pewno późniejsza niż pliku wynikowego). Reguła ta stosowana jest rekurencyjnie do wszystkich plików występujących po stronie zależności, powodując rekompilację niektórych źródeł, aż do zaspokojenia wszystkich zależności.

Przykładowo, jeżeli w pliku *Makefile* znajdą się następujące definicje:

```
test2.o: test2.c test.h
        $(CC) $(CFLAGS) -c -o $@ $<

test1.o: test1.c test.h
        $(CC) $(CFLAGS) -c -o $@ $<

# Plik test tworzymy z dwóch plików źródłowych, linkując je razem.
test: test1.o test2.o
        $(CC) $(LD_FLAGS) -o $@ $^
```

to występować będą następujące zależności[7] (sprawdzane po napisaniu `make test`):

- Modyfikacja pliku *test2.c* spowoduje konieczność rekompilacji pliku *test2.o*, a co za tym idzie, także pliku *test*. Plik *test1.o* nie będzie ruszany, gdyż w żadnym stopniu nie zależy on od modyfikacji dokonywanych w pliku *test2.c* ani *test2.o*.
- Modyfikacja pliku *test.h* wymusi rekompilację zarówno pliku *test2.o*, jak i *test1.o*, a więc w rezultacie przebudowany zostanie także plik *test*.
- Usunięcie pliku *test1.o* z dysku spowoduje konieczność kompilacji pliku *test1.c* w celu utworzenia *test1.o*. Po dokonaniu kompilacji plik *test1.o* będzie nowszy niż plik *test*, co z kolei wymusi wywołanie programu `cc` w celu zlinkowania pliku *test*.
- Usunięcie pliku *test* spowoduje wyłącznie konieczność ponownego zlinkowania tego pliku, nie ruszając plików *test1.o* ani *test2.o*.

Wpisanie `make` bez żadnych argumentów powoduje sprawdzenie zależności i ewentualne przebudowanie pierwszego celu, jaki jest zdefiniowany w pliku *Makefile*. W powyższym przykładzie byłby to plik *test2.o*,

[7] Użyte w powyższych przykładach znaki specjalne oznaczają odpowiednio: `$@` – nazwa celu występującego po lewej stronie zależności (czyli np. *test2.o* w pierwszym przykładzie), `$<` – pierwszy z plików występujących po prawej stronie zależności, czyli np. *test2.c* lub *test1.c*, `$^` – wszystkie pliki wymienione po prawej stronie zależności, czyli *test1.o* i *test2.o* w ostatnim przykładzie.

co raczej nie jest pożądanym zachowaniem, gdyż „ostatecznym" działaniem programu `make` powinno być utworzenie programu wykonywalnego o nazwie *test*. Aby zapewnić takie działanie, definicja zależności pliku *test* powinna wystąpić jako pierwsza, przed definicją zależności plików *test2.o* i *test1.o*. Często jednak stosuje się inne rozwiązanie, definiując pseudo-cel lub zależność o nazwie `all`, wpisując jako pierwszą definicję (z reguły za definicjami zmiennych takich jak `CFLAGS` i innych, ale koniecznie przed innymi zależnościami:

```
all: test
```

Zależność ta mówi, że w celu utworzenia czegoś o nazwie „all" należy zadbać o utworzenie aktualnej wersji pliku *test*, którego zależności znajdują się w dalszej części pliku *Makefile*. „all" jest tradycyjnie przyjętą nazwą oznaczającą zbudowanie wszystkich programów zdefiniowanych w *Makefile*, tzn. pisząc `make` lub `make all` mamy na myśli utworzenie skompilowanej wersji wszystkich programów, które są potrzebne. Jeśli na przykład w skład aplikacji sieciowej wchodzą dwa osobne programy – klient i serwer, które komunikują się ze sobą, a więc w celu przetestowania ich musimy najpierw uruchomić serwer, po czym połączyć się z nim za pomocą klienta, plik *Makefile* służący do ich utworzenia może wyglądać następująco:

```
CC=gcc
LIBS= -lsocket -lnsl
CFLAGS= -Wall -g

all: klient serwer

klient: klient.o wspolne.o socket.o

serwer: serwer.o wspolne.o socket.o

test: test.o socket.o

clean:
    rm -f *.o test klient serwer core core.*

zip: sso.zip

sso.zip: *.c *.h Makefile
    zip $@ $^
```

Napisanie `make` lub `make all` spowoduje utworzenie aktualnych wersji programów `klient` i `serwer`, natomiast utworzenie aktualnej wersji programu `test` wymaga napisania `make test`. Dodatkowo został zdefiniowany cel o nazwie „`clean`", nie mający żadnych zależności, czyli wykonywany zawsze po napisaniu `make clean`, a powodujący wykonanie komendy

`rm` usuwającej wszystkie pliki wykonywalne, pliki pośrednie i plik *core*, który może powstać gdy któryś z programów będzie próbował wykonać niedozwoloną operację, np. przy pomocy niewłaściwie zainicjowanego wskaźnika sięgając do losowo wybranego miejsca pamięci.

Ostatni cel – *sso.zip* pokazuje, jak użyć programu `make` do wykonywania innych, często powtarzanych czynności. Zdefiniowana zależność mówi, że plik *sso.zip* zależy od wszystkich plików z rozszerzeniami *.c* i *.h*, a także od samego pliku *Makefile*. Jeśli którykolwiek z nich został uaktualniony, trzeba przebudować plik *sso.zip*, a metoda przebudowania określoina jest w następnym wierszu: należy uruchomić program `zip` z odpowiednimi parametrami. Pierwszym z nich jest nazwa archiwum (czyli *sso.zip* podstawiane w miejsce `$@`), a potem nazwy wszystkich plików umieszczanych w archiwum (czyli lista zależności po prawej stronie dwukropka, podstawiana w miejsce `$^`). Dodatkowo został zdefiniowany pseudo-cel o nazwie `zip`, aby zamiast `make sso.zip` można było napisać `make zip`.

# 5 Debugowanie programów za pomocą gdb

Podstawowym debuggerem w systemach UNIX jest GNU Debugger, czyli `gdb`. Jest to debugger działający w trybie liniowym, istnieją jednak różne nakładki na niego, takie jak `xxgdb` itp., pozwalające na pracę w trybie pełnoekranowym. Oprócz tego w samym debuggerze istnieje wiele ułatwień pozwalających na wygodną pracę, takich jak automatyczne wyświetlanie zmiennych, pułapki jednorazowe, pomoc kontekstowa, kontekstowe rozwijanie nazw komend i zmiennyh, rozpoznawanie typów danych języków wysokiego poziomu, itp.

Kilka cech programu `gdb`:

- Możliwość debugowania dowolnych programów, nawet bez znajomości ich kodu źródłowego
- Możliwość podłączania się z debuggerem do już działającego programu
- Możliwość uruchomienia kross-debuggera, tzn. możliwość wykorzystania debuggera przez emulatory innych procesorów w celu debugowania emulowanych programów (np. emulator PalmOS wykorzystujący `gdb-m68k-coff` do debugowania w systemie UNIX aplikacji PalmOS na procesor Motorola 68030)
- Możliwość zdalnego debugowania aplikacji przez sieć (linię PTY) lub port szeregowy.

Aby w pełni wykorzystać możliwości debuggera, program pisany w C lub C++ powinien być skompilowany z opcją `-g` lub `-ggdb`:

```
gcc -o program -ggdb -Wall program.c
gdb program
```

## 5.1 Przykład debugowania programu

Aby poznać możliwości debuggera spróbujemy prześledzić wykonanie następującego programu:

```
 1 #include <stdio.h>
 2
 3 typedef struct msg {
 4   int id;
 5   char buf[100];
 6  } Msg;
 7
 8 int i=5;
 9
10 int parzysta(int x)
11 {
12   if (x%2)
13     return 1;
14   else
15     return 0;
16 }
17
18 int main (int argc, char ** argv)
19 {
20   int i,a,b[10], *p;
21   Msg msg[10], *mptr;
22
23   for (i=0, p=b, mptr=msg+10; i<10 && i<argc; i++) {
24     *p++=parzysta(i);
25     (*--mptr).id=i;
26     snprintf(mptr->buf,100, "%04x: %s", i, argv[i]);
27      printf("%d: %d, %s\n", i, b[i], mptr->buf);
28   }
29   return 0;
30  }
```

Program należy skompilować z opcją `-g` lub `-ggdb`, najlepiej dopisując ją do zmiennej `CFLAGS` w pliku *Makefile*.

Po uruchomieniu `gdb program` na początek zostaje wczytana tablica symboli programu, ale program nie zostaje uruchomiony, pozostawiając nam inwencję co tak naprawdę chcemy z programem zrobić. Możemy uruchomić program wpisując `run` lub `run lista argumentów`. To spowoduje jednak uruchomienie całego programu i jego zakończenie po wykonaniu, dlatego z reguły pierwszą czynnością po uruchomieniu gdb jest wpisanie `break main`, aby ustawić pułapkę na początku funkcji *main*().

Pułapki można ustawiać na kilka sposobów:

- `break` *nazwa funkcji*

  ustawia pułapkę na początku podanej funkcji. W przypadku programów w C++ często stosowaną składnią będzie *NazwaKlasy.funkcja*.

- `break` *numer linii*

  ustawia pułapkę w podanym wierszu aktualnie debugowanego programu źródłowego (tego samego, w którym znajduje się aktualnie wykonywana funkcja)

- `break` *plik.c:nrlinii*

  ustawia pułapkę w podanym wierszu innego pliku.

- `tbreak ...`

  ustawia pułapkę tymczasową, tzn. po pierwszym trafieniu zostaje ona automatycznie usunięta.

Jeśli ustawiliśmy jakąś pułapkę (np. `break main` lub `break 16`), możemy rozpocząć wykonywanie programu. Najprościej – pisząc `run`, a jeśli chcemy do programu przekazać jakieś argumenty, to np. `run abc def -z 1`.

Po uruchomieniu program natychmiast zostaje zatrzymany, a gdb informuje nas w którym miejscu:

```
GNU gdb (5.1.90CVS-5)
(gdb) break main
Breakpoint 1 at 0x8048455: file pg.c, line 23.
(gdb) run abc def -z 1
Starting program: ./pg abc def -z 1

Breakpoint 1, main (argc=5, argv=0xbffff664) at pg.c:23
23   for (i=0, p=b, mptr=msg+10; i<10 && i<argc; i++) {
(gdb)
```

To oznacza, że `gdb` zatrzymał program tuż przed wykonaniem wyświetlonej linii. Możemy się o tym łatwo przekonać pisząc na przykład:

```
(gdb) p i
$1 = 134513681
(gdb) n
24     *p++=parzysta(i);
(gdb) print i
$2 = 0
(gdb)
```

`print` lub w skrócie `p` pozwala wydrukować wartość zmiennej, widać więc, że zmienna `i` nie jest jeszcze zainicjowana i zawiera „śmieci" - dopiero gdy wpiszemy `next` lub w skrócie `n`, spowoduje to wykonanie pojedynczej instrukcji programu, a więc wykona się instrukcja `for` i zaraz potem po sprawdzeniu wartości `i` przekonamy się, że jest to 0.

Jeśli wykonamy kolejną instrukcję, będziemy mogli sprawdzić także jaka wartość została zapisana w tablicy `b`:

```
(gdb) n
25      (*--mptr).id=i;
(gdb) p p
$6 = (int *) 0xbffff5b4
(gdb) p *p
$7 = 1073787046
(gdb) p *(p-1)
$8 = 0
(gdb)
```

Wydrukowanie wartości `p` jest mało interesujące, gdyż jest to wskaźnik, dowiadujemy się więc na jaki adres pamięci wskazuje. Interesuje nas raczej zawartość komórki wskazywanej przez `p`, dlatego wpisaliśmy `p *p`. To też nie jest dobre w tym przypadku, bo z powodu użycia operatora `++` zmienna `p` wskazuje już na kolejny element tablicy, dlatego wpisując `*(p-1)` w końcu otrzymujemy zawartość komórki poprzednio wskazywanej przez `p`.

`gdb` pozwala na korzystanie z arytmetyki wyrażeń tak samo jak język, w którym napisany jest debugowany program, czyli w tym przypadku C. Zadziałają więc operatory takie jak `->` czy `*`, zmiany typów, itp. itd. Wszystkie podawane nazwy zmiennych czy funkcji dotyczą zawsze lokalnego kontekstu, czyli tak samo, jak by je traktował kompilator. Można się jednak odwołać „wyżej", podając wybrany kontekst przed podwójnym dwukropkiem, np:

```
(gdb) p i
$9 = 0
(gdb) p main::i
$10 = 0
(gdb) p ::i
$11 = 5
(gdb)
```

Jak widać zmienna `i` w kontekście funkcji *main()*, czyli jej zmienna lokalna, ma wartość 0, a globalna zmienna `i` – wartość 5.

Wykonywanie kodu można kontynuować na jeden z kilku sposobów:

- `next`, `n`

  Wykonuje kolejną pełną linię programu. Jeśli następuje wywołanie funkcji, nie jest śledzony przebieg programu w tej funkcji.

- `step`, `s`

  Wykonuje kolejną pełną linię programu, pokazując także krok po kroku wykonanie funkcji.

- `continue`, `c`, `cont`

  Wznawia wykonanie programu. Program będzie się wykonywał samodzielnie, aż do omomentu gdy się zakończy, natrafi na ustawioną wcześniej pułapkę, albo otrzyma sygnał (np. `SIGINT` spowodowany naciśnięciem `Ctrl-C`).

- `return`, `ret`

  Wewnątrz funkcji – wznawia wykonanie programu (jak `cont`, ale nie dłużej niż do opuszczenia aktualnej ramki stosu, czyli wyjścia programu z funkcji.

- `stepi`, `si`, `nexti`, `ni`

  Jak `next` czy `step`, ale dla dokładnie jednej instrukcji asemblera.

W trakcie działania programu do orientowania się o jego stanie możemy użyć komend:

- `list`, `l`

  listuje kilka linii programu naokoło aktualnie wykonywanej.

- `backtrace`, `bt`

  pokazuje zawartość stosu, czyli nazwy wołanych kolejno funkcji, wraz z wartościami przekazanych zmiennych, na przykład:

```
(gdb) tbreak parzysta
Breakpoint 3 at 0x8048433: file pg.c, line 12.
(gdb) run abc def -z 1
Starting program: pg abc def -z 1

Breakpoint 1, parzysta (x=0) at pg.c:12
12   if (x%2)
(gdb) bt
#0  parzysta (x=0) at pg1.c:12
#1  0x08048497 in main (argc=5, argv=0xbffff664) at pg1.c:24
#2  0x42017499 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

- `print`, `p`
  `printf`

  pozwalają wyświetlić wartość zmiennej

- `display`, `disp`

  pozwala automatycznie wyświetlać podaną wartość przy każdym zatrzymaniu programu.

Inne polecenia `gdb`:

- `attach` *`pid`*

  Przyłącza się do procesu o podanym numerze (uruchomionego wcześniej, niezależnie od `gdb`), zatrzymuje go i pozwala debugować.

- `delete`, `del`

  Usuwa wskazaną pułapkę, automatyczne wyświetlanie zmiennej itp. Numer pułapki do usunięcia można uzyskać komendą `info break`.

- `info`

  Wyświetla różnego rodzaju informacje dotyczące aktualnej sesji pracy z `gdb`, np. na temat pułapek, numeru procesu, itp.

- `help`

  Przywołuje pomoc dotyczącą poszczególnych komend.

Przy okazji warto wspomnieć o debugowaniu programów wykonujących *fork*(). Po wykonaniu tej funkcji zamiast jednego procesu dostajemy dwa – rodzica i dziecko. Jeśli proces wykonujący tę funkcję robił to pod kontrolą `gdb`, to kontrolowany pozostaje zawsze proces rodzica, a dziecko wykonuje się dalej samodzielnie, bez udziału `gdb`. Jeśli chcemy śledzić także to, co dzieje się w dziecku, należy postępować następująco:

- W kodzie programu dziecka jedną z pierwszych instrukcji po wykonaniu *fork*() powinno być *sleep*(15) lub wywołanie tej funkcji z inną wartością (czas w sekundach), dającą nam dość czasu, by wykonać kolejne czynności opisane poniżej.

- Po uruchomieniu `gdb program` ustawiamy pułapkę za funkcją *fork*(), w kodzie rodzica. Możemy także zawczasu w drugim oknie/terminalu uruchomić `gdb program`, który za chwilę stanie się debugowanym procesem-dzieckiem. W pierwszym oknie uruchamiamy program komendą `run`.

- Po zatrzymaniu programu sprawdzamy numer procesu potomnego – komendą `ps` (lub badając w `gdb` zawartość zmiennej, do której zapisaliśmy wartość zwróconą przez *fork*()).

- W oknie z `gdb` przygotowanym do debugowania procesu-dziecka wpisujemy `attach` *`numer-procesu-dziecka`*. Spowoduje to przejęcie przez `gdb` kontroli nad śpiącym jak dotąd w funkcji *sleep*() dzieckiem i wstrzymanie jego wykonywania. Od tego momentu możemy

proces dziecka uruchamiać w trybie krokowym, ustawiać pułapki, badać zmienne, itd.

# 6 Ćwiczenia projektowe

Zamieszczone poniżej tematy proszę traktować jako uzupełnienie informacji, które będą zamieszczane przed zajęciami na ePortalu i na stronie WWW przedmiotu – tam będzie podany szczegółowy zakres ćwiczenia i wymagania. Poniższe zestawienie ma charakter informacyjny, a lista ćwiczeń lub ich zakres może ulec zmianie.

## 6.1 Wykorzystanie funkcji *fork()* i *pipe()*

Celem ćwiczenia jest napisanie programu, który utworzy dwa procesy komunikujące się ze sobą za pomocą jednokierunkowego strumienia danych (ang. *pipe*). Strumień taki utworzyć można przy pomocy funkcji *pipe()*. Funkcja ta zwraca w postaci tablicy 2 deskryptory, które są ze sobą połączone strumieniem danych, tzn. cokolwiek zostanie zapisane do jednego z tych deskryptorów, może zostać odczytane z drugiego[8].

Użycie funkcji *fork()* powoduje utworzenie dodatkowego procesu – a dokładniej: rozdzielenie się aktualnie wykonywanego procesu na dwa podprocesy. Po wykonaniu funkcji *fork()* sterowanie w obu procesach przebiega tak, jakby nic się nie stało, tzn. oba procesy zawierają w pamięci obraz tego samego programu, mają dostęp do tych samych danych, mają otwarte te same deskryptory plików i są kontynuowane od instrukcji znajdującej się za wywołaniem funkcji *fork()*. Jednak jeden z nich jest „rodzicem", a drugi „dzieckiem", a można je rozróżnić po tym, jaką wartość zwróciła funkcja *fork()*.

Ćwiczenie projektowe składa się z dwóch części.

W części pierwszej (1a) powinien zostać napisany program, który będzie tworzyć w procesie głównym strumień danych, po czym podzieli się na dwa procesy, korzystając następnie z tego strumienia komunikacyjnego do przesłania danych – np. tekstu wczytanego z klawiatury lub dowolnych innych informacji, wyświetlonych następnie w drugim z procesów.

Dodatkowo (1b), przetestować należy pojemność strumienia pipe, poprzez takie pokierowanie procesami (np. z wykorzystaniem funkcji *sleep()*), aby proces odbierający dane został na chwilę wstrzymany, podczas

[8] w większości systemów UNIX, w tym także w systemie Linux, każdy „koniec" ma swoje ścisłe przeznaczenie, tzn. nie jest obojętne który deskryptor jest używany do zapisywania informacji, a który do odczytywania. Szczegółowe informacje na ten temat znajdują się na stronach dokumentacji systemowej (komenda `man`) oraz [4].

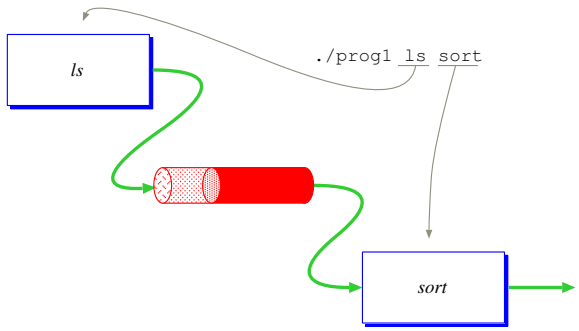
gdy proces wysyłający dane zapisuje je nieprzerwanym strumieniem.

W drugiej części ćwiczenia (2a) należy zastąpić jeden z procesów jakimś programem systemowym (np. `ls` oraz `sort` albo `cat -n`) podawanym z linii komend i wywołanymi za pomocą którejś funkcji z rodziny *exec*(), łącząc wyjście pierwszego procesu z wejściem drugiego.

Jako zadanie dodatkowe (2b) należy wykonać mini-shella, czyli program, któremu jako parametry podajemy nazwy programów do uruchomienia, a on je tworzy jako podprocesy połączone strumieniem pipe. Na przykład uruchomienie:

```
./minishell ls cat
```

spowoduje uruchomienie podprocesu zastąpionego za chwilę programem `ls`, który dane wyprodukowane na swoje wyjście przekaże na wejście programu `cat` działającego jako drugi podproces, tworząc 2 procesy połączone w sposób przedstawiony na rysunku poniżej:




Przekazywanie parametrów do podprocesów można na początek ustaić w sposób statyczny, np. wymagać podania 1 nazwy programu i 1 argumentu dla niego.

Dla ambitnych (2c): minishell nie powinien być ograniczony co do liczby argumentów, czyli można go uruchomić dla 2, 5 albo 10 programów połączonych strumieniami pipe. Można przyjąć, że parametry oznaczają na zmianę nazwę programu i jego parametr (i jest ich wtedy wymagana parzysta liczba), albo np. ustalić znak separatora (odpowiednik znaku `|`) – na przykład kropkę. Wówczas uruchomienie kilku procesów mogłoby wyglądać tak:

```
./minishell ls -l . cat -n . sort -k2 -r . less
```

Szczególną uwagę podczas pisania i testowania programów należy zwrócić na:

- sprawdzanie wartości zwracanych przez funkcje systemowe i reagowanie na sygnalizowane w ten sposób błędy występujące w trakcie wykonywania programu;
- włączenie podczas kompilacji opcji drukowania wszystkich ostrzeżeń, pozwalające uniknąć większości błędów powstających na etapie pisania programu;
- wywoływanie odpowiednich funkcji systemowych powodujących zwalnianie uprzednio przydzielonych zasobów i synchronizację wykonywania obu procesów;
- czytelne formatowanie treści programu.

## 6.2 Użycie nazwanych strumieni (*named pipes*)

Oprócz strumieni nienazwanych (tworzonych funkcją *pipe*()) w systemie UNIX można korzystać także ze strumieni nazwanych. Tworzone są one za pomocą funkcji *mkfifo*() lub *mknod*() (albo z linii komend – za pomocą komend `mkfifo` lub `mknod`). Powodują one utworzenie w systemie plików pliku o zadanej nazwie, który jest jednak plikiem specjalnym, co widać choćby oglądając prawa dostępu do tego pliku. Funkcja *mknod*() może służyć także do tworzenia urządzeń specjalnych (blokowych lub znakowych), ale do tego wymagane są uprawnienia administratora systemu, dlatego zalecanym sposobem postępowania w przypadku strumieni FIFO jest korzystanie z *mkfifo*, które jest całkowicie bezpieczne i przenośne pomiędzy różnymi systemami.

Procesy komunikujące się za pomocą tak utworzonego strumienia danych nie muszą być ze sobą w żaden sposób spokrewnione, gdyż nie są pomiędzy nimi przekazywane żadne deskryptory otwartych plików. Nawiązanie połączenia polega na otwarciu do zapisu lub do odczytu pewnego „z góry ustalonego” pliku specjalnego.

Oprócz pokazania dwóch (osobno uruchamianych) procesów przekazujących dane należy przetestować sytuacje, w których:

- Proces odbierający ze strumienia symuluje ich powolne odbieranie, np. wprowadzając 15-30-sekundowe opóźnienie w procesie odbierającym, podczas gdy proces wysyłający dane zapisuje je nieprzerwanym strumieniem;
- Proces piszący jest jedynym procesem łączącym się do strumienia (brak odbiorców, tzn. nie ma jeszcze żadnego procesu – czytnika);
- Proces piszący zasypia na jakiś czas, podczas gdy proces odbierający jest gotów na przyjmowanie informacji;

Jaka jest pojemność takiego strumienia?

Czy strumień danych utworzony w ten sposób może egzystować w systemie dłużej niż program, który go utworzył?

W drugiej części laboratorium należy napisać program, który stanie się front-endem do programu `mplayer`, pozwalając na odmienne jego sterowanie. Program `mplayer` należy uruchomić z opcjami:

`mplayer -input file=`*/tmp/StrumieńFifo plik.mp3*

tworząc wcześniej podany strumień FIFO programowo lub z linii komend. Napisany program powinien interpretować pojedyncze litery ze standardowego wejścia (albo litera+znak Enter), przekładać je na odpowiednie komendy mplayera i wysyłać je do tego strumienia (np. „`seek 3`", „`seek -5`", „`pause`", itp.). Listę rozpoznawanych komend można sprawdzić w manualu programu `mplayer`

Szczególną uwagę podczas pisania i testowania programów należy zwrócić na:

- sprawdzanie wartości zwracanych przez funkcje systemowe i reagowanie na sygnalizowane w ten sposób błędy występujące w trakcie wykonywania programu;
- właściwe prawa dostępu do tworzonego strumienia danych, umożliwiające co najmniej dostęp do tego strumienia jego właścicielowi (czyli procesom uruchamianym z tym samym *uid*, co proces tworzący strumień;
- konieczność zwalniania zasobów przydzielonych podczas działania programu (funkcja *unlink*() i komenda `rm`);
- synchronizację procesów między sobą.

*UWAGA!* – strumienie najlepiej jest tworzyć na lokalnym dysku komputera, a więc na przykład w katalogu */tmp*, a nie we własnym katalogu domowym, który jest podłączany przez NFS z serwera. Bie wszystkie wersje protokołów NFS obsługują poprawnie urządzenia specjalne, jakim są strumienie FIFO, sporo też zależy od lokalnie używanych opcji montowania dysków sieciowych. W laboratoriach 103/104/127 C-3 akurat nie powinno to być problemem, ale i tak bezpieczniej jest robić to w katalogu *tmp*.

Zamiast pisać 2 niezależne programy można napisać jeden, uruchamiany z parametrem (dostępnym przez `argc` i `argv`) i zależnie od niego różnicujący swoje działanie.

## 6.3 Semafory i pamięć wspólna

Semafory są mechanizmem programowania współbieżnego służącym do synchronizacji działania procesów. Można ich także użyć do przekazywania informacji pomiędzy procesami, lecz w bardzo niewielkim zakresie – ograniczając się do informacji, że jeden proces osiągnął pewien punkt wykonania, co oznacza, że inny proces, który jest uzależniony od wykonania sygnalizowanych w ten sposób działań, może kontynuować swoje wykonywanie. Podstawowym zastosowaniem pozostaje jednak ograniczanie równoczesnego dostępu kilku procesów do wspólnych danych lub wykluczenie jednoczesnego wykonywania przez kilka procesów pewnych operacji.

System operacyjny UNIX dostarcza semaforów *ogólnych*[2], tzn. mogących przyjmować dowolne wartości nieujemne. Semafory *binarne*, czyli przyjmujące wyłącznie wartości 0 lub 1 są podzbiorem semaforów ogólnych, a więc także można z nich korzystać, nadając semaforowi odpowiednią wartość początkową przy użyciu funkcji *semctl*(). Dodatkowo, semafory w systemie UNIX tak naprawdę są grupami semaforów (szczególnym przypadkiem jest grupa, na którą składa się dokładnie jeden semafor), a operacje na grupie semaforów są wykonywane w niepodzielny sposób. Można więc na przykład w jednej operacji zażądać zajęcia kilku semaforów i operacja ta uda się tylko wtedy, gdy możliwe będzie zajęcie wszystkich wymienionych semaforów.

Możliwe jest także użycie semaforów w trybie bez blokady procesu, jeżeli zajęcie semafora jest niemożliwe – proces nie jest wówczas zawieszany, lecz funkcja *semop*() zwraca wartość oznaczającą błąd, program jest kontynuowany i ma możliwość wykonania innych operacji przed ponowną próbą zajęcia semafora. W ten sposób jednak nie ma gwarancji uzyskania dostępu do semafora, gdyż może się zdarzyć, że zawsze w chwili podjęcia takiej próby jest on zajęty. W „normalnym" (blokującym) wywołaniu funkcji *semop*(), jeśli semafor nie jest dostępny, proces jest umieszczany w kolejce procesów oczekujących na zwolnienie semafora, a kolejka obsługiwana jest w takiej kolejności, w jakiej procesy były do niej dopisywane.

Pamięć współdzielona pozwala pokonać barierę stawianą pomiędzy osobnymi procesami – nawet procesy tworzone za pomocą funkcji *fork*() w momencie podziału uzyskują osobne segmenty pamięci przeznaczone na dane, więc nie mogą nawzajem ingerować w swoje zmienne. Jedyną wspólną rzeczą, które dziedziczą są deskryptory otwartych plików (gdyż są to tak naprawdę wskaźniki na struktury znajdujące się w jądrze systemu i wspólne dla wszystkich procesów). czasem jednak przydatny jest blok pamięci, który byłby widoczny przez kilka procesów jednocześnie i służący im do wymiany informacji (przy zachowaniu odpowiedniej synchronizacji).

W tym celu można użyć pamięci współdzielonej (*shared memory*). Podobnie jak w przypadku kolejek komunikatów czy nazwanych strumieni, aby dostać się do pamięci współdzielonej procesy muszą uzgodnić lub znać wcześniej pewną nazwę służącą do utworzenia klucza dostępu. W przeciwnym razie zamiast korzystać z tego samego bloku pamięci współdzielonej będą próbowały korzystać z rozłącznych bloków pamięci i żadna komunikacja nie będzie miała miejsca. Podobnie jak w innych mechanizmach komunikacji międzyzadaniowej tylko jeden proces powinien tworzyć blok pamięci, do czego służy funkcja *shmget*(). Natomiast każdy z procesów, który chce korzystać z pamięci współdzielonej musi korzystać z funkcji *shmat*() i *shmdt*(), które powodują „zakotwiczenie" takiego bloku w przestrzeni adresowej procesu oraz zwolnienie bloku, gdy jest on już niepotrzebny.

Omówienie zastosowań pamięci współdzielonej można znaleźć w [3], natomiast sposób jej wykorzystania najlepiej opisany jest w [4].

Celem ćwiczenia jest zaprezentowanie działania semaforów w celu ograniczenia dostępu kilku procesów do wspólnej sekcji krytycznej. Należy w tym celu ) napisać program działający według scenariusza, w którym kilka procesów próbuje kilkakrotnie wejść do sekcji krytycznej, np. w celu modyfikacji pewnej wspólnej zmiennej (tworzonej z użyciem pamięci współdzielonej).

Proponowany scenariusz – konto bankowe, z podziałem na poszczególne etapy:

- Kilka procesów (uruchamianych funkcją *fork*() lub ze skryptu shell) konkuruje o dostęp do komórki w pamięci wspólnej – konta bankowego. Należy uruchomić kilka (2-10) procesów dokonujących odpowiednio wpłat i wypłat, ale w przewidywalny sposób – aby można było wyliczyć, jakie powinno być końcowe saldo konta, np. przy 5 procesach wpłacających 10 razy po 100 jednostek i 5 procesach, z których każdy 10 razy wypłaca 100 jednostek końcowe saldo powinno wynosić zero (ale można przyjąć dowolny inny scenariusz z predefiniowaną liczbą wpłat/wypłat o ustalonej z góry wartości). Najprostsze może być uruchamianie procesów z powłoki – czyli przekazywanie parametrów za pomocą `argc` i `argv` – np. uruchomienie procesu z parametrami `konto 10 100` może oznaczać wykonanie w pętli 10 wpłat o wartości 100, a `konto 15 -100` – dokonanie 15 wypłat o wartości 100 każda. Kwestię tworzenia mechanizmów pamięci wspólnej i semaforów można rozwiązać analogicznie – uruchamiając program z odpowiednimi parametrami. Przy takim podejściu do zagadnienia, skrypt startowy powinien utworzyć mechanizmy pamięci wspólnej i semaforów, następnie uruchomić kilka-kilkanaście równolegle działających procesów wpłat/wypłat, a po ich zakończeniu – „posprzątać" – kasując utworzone mechanizmy (można to zrobić nawet z poziomu shell-a, za

pomocą programów takich jak `ipcs` i `ipcrm`.

- Etap pierwszy – dostęp do pamięci wspólnej (konta) bez mechanizmów synchronizacji – wynik oczywiście może być inny od spodziewanego ze względu na gubienie niektórych operacji
- Etap drugi – dodanie mechanizmu synchronizacji/ochrony sekcji krytycznej za pomocą semaforów IPC. Po włączeniu mechanizmu i przy jego poprawnym działaniu saldo końcowe powinno być zawsze takie, jakiego się spodziewamy.
- Etap trzeci (opcjonalny) – dwa konta zamiast jednego i dodatkowe procesy dokonujące także przelewów pomiędzy kontami

W celu najlepszego zobrazowania tego, co się dzieje, przed i po wywołaniach funkcji zajmujących i zwalniających semafor powinny znaleźć się wywołania funkcji *printf*(), drukującej na standardowe wyjście informacje o tym, co dany proces zrobił lub zamierza zrobić (np. próbuje zająć semafor lub informuje, że mu się to właśnie udało).

Inny sposób wykonania programu to utworzenie w pamięci współdzielonej bufor danych wykorzystywanego przez kilka procesów – jedne z nich zapisują do bufora pewne informacje, inne odczytują je i drukują na standardowym wyjściu, korzystając z semaforów w celu synchronizacji dostępu.

Szczególną uwagę należy zwrócić na sposób synchronizacji procesów, tzn. sposób przekazania informacji, że pierwszy proces zapisał informacje do bufora, wobec czego drugi proces może je z tego bufora odczytać.

## 6.4 Sygnały

Sygnały służą z reguły do sterowania procesami w sytuacjach wyjątkowych, takich jak np. przekazanie informacji o zakończeniu działania innego procesu, wymuszenie zakończenia procesu przez inny proces, lub poinformowanie procesu przez system operacyjny o wystąpieniu jakiejś sytuacji specjalnej. Przykładem sygnałów wysyłanych przez system operacyjny mogą być:

- `SIG_PIPE` wysyłany do procesu, który uprzednio wysłał dane do strumienia pipe lub FIFO, a strumień ten został później zamknięty bez odczytywania tych danych.
- `SIG_WINCH` wysyłany do procesów działających w oknach X11, gdy zmieni się rozmiar okienka związanego z procesem lub jego terminalem kontrolnym.

- SIG_INT powodujący przerwanie procesu, wysyłany zwykle gdy użytkownik naciśnie klawisz Ctrl-C.
- SIG_SUSP wysyłany, gdy proces zostaje wstrzymany (naciśnięcie przez użytkownika Ctrl-Z).
- SIG_CLD wysyłany przez system w przypadku zakończenia działania któregoś z procesów potomnych.

Niektóre z tych sygnałów są domyślnie ignorowane (np. SIG_CLD lub SIG_WINCH, inne powodują standardowo przerwanie procesu (np. SIG_PIPE lub SIG_INT). Wszystkie, z wyjątkiem SIG_KILL i SIG_STOP mogą być przechwycone.

Z poziomu linii komend do wysłania sygnału można użyć komendy kill, a z poziomu programu w języku C – funkcji *kill*(). Do „własnych" zastosowań przewidziano sygnały SIG_USR1 i SIG_USR2.

Celem ćwiczenia jest napisanie programu pokazującego zachowanie programu przechwytującego sygnały. Należy uodpornić program przynajmniej na przerwanie go przez użytkownika przez naciśnięcie Ctrl-C (program powinien wówczas wypisać na ekranie komunikat o przechwyceniu sygnału, zamknąć wszystkie otwarte pliki i dopiero zakończyć działanie) oraz przedwczesne zamknięcie strumienia przez proces czytający (w tym celu można wykorzystać fragmenty programu pokazującego działanie strumienia pipe lub FIFO – w procesie głównym zapisać pewną ilość danych do strumienia, a w drugim procesie, po odczytaniu kilku bajtów ze strumienia zamknąć go, powodując w ten sposób wysłanie sygnału SIG_PIPE do procesu głównego. Należy też pokazać wysyłanie sygnałów za pomocą *kill*() (np. wysyłając sygnał SIG_USR1 nie przerywający działania programu, lecz jedynie drukujący informację o otrzymaniu sygnału.

# Literatura

[1] Maurice J. Bach. *Budowa systemu operacyjnego UNIX*. Wydawnictwa Naukowo-Techniczne, 2 edition, 1995. oryginał: "The Design of the UNIX Operating System", Prentice Hall, Inc., 1986.

[2] M. Ben-Ari. *Podstawy programowania współbieżnego*. Wydawnictwa Naukowo-Techniczne, 1 edition, 1989. oryginał: "Principles of Concurrent Programming", Prentice-Hall International (UK) Ltd., 1983.

[3] Abraham Silberschats, James L. Peterson, and Peter B. Galvin. *Podstawy systemów operacyjnych*. Wydawnictwa Naukowo-Techniczne, 2 edition, 1993. oryginał: "Operating System Concepts", Addison-Wesley Publishing Company, Inc., 1991.

[4] W. Richard Stevens. *Programowanie zastosowań sieciowych w systemie Unix*. Wydawnictwa Naukowo-Techniczne, 1 edition, 1995. oryginał: "UNIX Network Programming", Prentice Hall, Inc., 1990.

# Spis treści