Programowanie w ANSI C z wykorzystaniem funkcji jądra systemu UNIX/Linux

Programowanie Współbieżne (dawniej Systemy Operacyjne II)

Tadeusz Kobus, Maciej Kokociński
Instytut Informatyki, Politechnika Poznańska

Organizacja – studia stacjonarne

Plan:

1. Operacje na deskryptorach
2. Procesy
3. Sygnały
4. Potoki, kolejki FIFO
5. Komunikacja między procesami i ich synchronizacja
6. Wątki POSIX

Zaliczenie:

- Zadania domowe
- 2 kolokwia
- Projekt

Organizacja – studia niestacjonarne

Plan:

1. Operacje na deskryptorach
2. Procesy
3. Potoki, kolejki FIFO
4. Wątki POSIX

Zaliczenie:

- Zadania domowe
- Kolokwium

Materiały

http://www.cs.put.poznan.pl/tkobus/
students/sop2/sop2.html

http://www.cs.put.poznan.pl/akobusinska/sop2.html
http://www.cs.put.poznan.pl/dwawrzyniak/
http://www.cs.put.poznan.pl/csobaniec/edu/sop/sop2.html

Język C – przypomnienie (1)

Kompilacja:

```
$ gcc program.c -o program
$ gcc -Wall program.c -o program
$ gcc -g program.c -o program
```

Uruchomienie:

```
$ ./program
```

Pomoc systemowa:

- sekcja 2 – funkcje systemowe (np. `man 2 write`)
- sekcja 3 – funkcje biblioteczne (np. `man 3 printf`):

Język C – przypomnienie (2)

```c
#include <stdio.h>

#define SIZE 10

int global = 4;

void main() {
    char c = 'c';
    int v = 42;
    float x = 3.14f;

    printf("%c %d %f\n", c, v, x);
    printf("%d %d\n", SIZE, global);

    int *p = &v;
    printf("%d == %d\n", v, *p);
}
```

Język C – przypomnienie (3)

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void main() {
    char chars[SIZE] = "Hello!\n";
    int values[4] = { 4, 2, 8, 9 };
    float *results = (float *)malloc(SIZE * sizeof(float));

    results[0] = 4.0f;
    results[1] = 2.2f;
    results[2] = 8.7f;
    results[3] = 9.1f;

    printf("%d == %d\n", values[2], *(values+2));
    printf("%f == %f\n", results[2], *(results+2));
}
```

Język C – przypomnienie (4)

```c
#include <stdio.h>
#include <string.h>
#define N 10

void main() {
    char chr1[N] = "Hello\n";
    char chr2[N] = "Hello\n\0";
    char chr3[N] = {'H', 'e', 'l', 'l', 'o', '\n', 0, 0, 0, 0};

    printf("%s%s%s", chr1, chr2, chr3);
    printf("%d %d %d\n", sizeof chr1, sizeof chr2,
        sizeof chr3);
    printf("%d %d %d\n", strlen(chr1), strlen(chr2),
        strlen(chr3));
}
```

Język C – przypomnienie (5)

```c
#include <stdio.h>
#include <string.h>
#define N 10

void main() {
    char chr1[N] = "Hello\n";
    char *chr2 = chr1;
    char *chr3 = "Hello\n";

    printf("%s%s%s", chr1, chr2, chr3);
    printf("%d %d %d\n", sizeof chr1, sizeof chr2,
        sizeof chr3);

    chr1 = chr3; // błąd
    strcpy(chr1, chr3);

    if (chr1 == chr2) printf("the same\n"); // błąd
    if (strcmp(chr1, chr2) == 0) printf("the same\n");
}
```

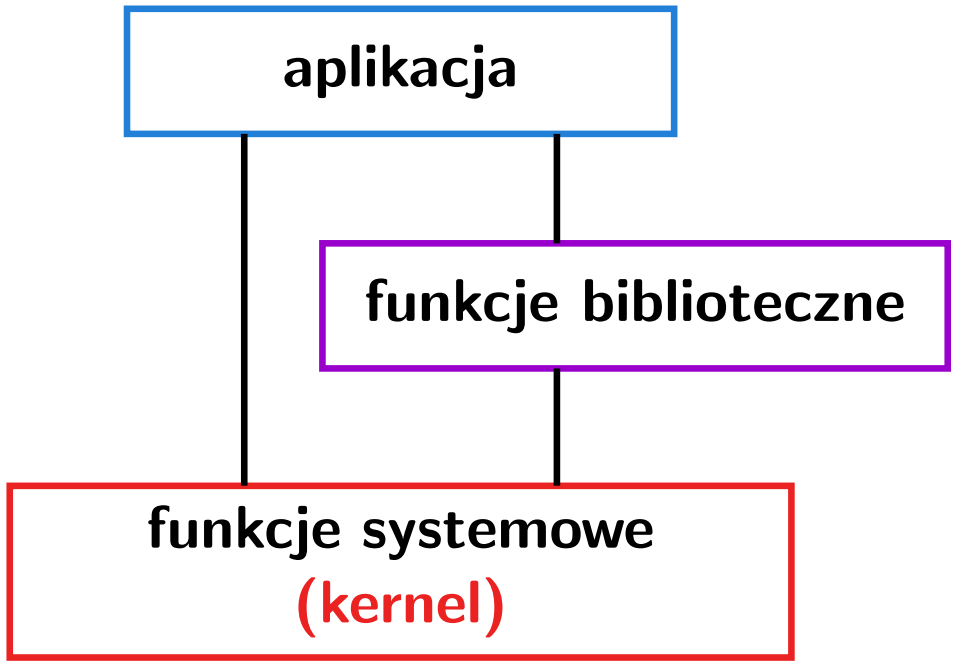
Język C – przypomnienie (6)

```c
#include <stdio.h>

int foo(int a, int *b) {
    int c;
    a++;
    (*b)++; // *b = *b + 1;
    c = a + *b;
    printf("%d + %d = %d\n", a, *b, c);
    return c;
}

void main() {
    int a = 2;
    int b = 3;
    int c = foo(a, &b);
    printf("%d + %d = %d\n", a, b, c);
}
```

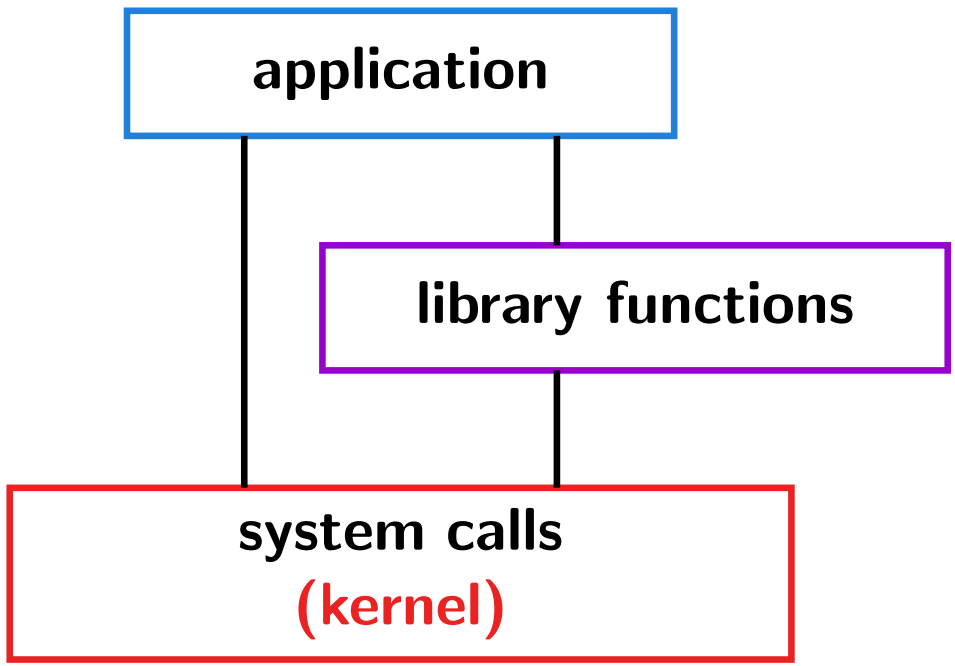
Język C – przypomnienie (7)

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int i;

  printf("number of arguments: %d\n", argc);

  for (i = 0; i < argc; i++)
      printf("argument %d: %s\n", i, argv[i]);

  return 0;
}
```

Architektura systemu operacyjnego



Architektura systemu operacyjnego



Funkcje systemowe (1)

Mechanizm komunikacji z jądrem systemu operacyjnego, który zarządza zasobami komputera.

Funkcje systemowe udostępnione są w formie bibliotek funkcji.

Funkcje systemowe wykonywane są w trybie jądra, tj. w uprzywilejowanym środowisku systemu operacyjnego, ale z uwzględnieniem ograniczeń dotyczących użytkownika uruchamiającego program.

Funkcje systemowe (2)

Ustandaryzowany interfejs POSIX (Portable Operating System Interface for Unix)

Bardzo często funkcja zwraca wartość typu `int`:

- $\geqslant 0$ – funkcja zakończyła się poprawnie
- -1 – funkcja zakończyła się błędnie; kod błędu ustawiany jest w zmiennej `errno`

Przykład:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Przykład

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int v = 2015;

    printf("%10d", v);
}
```

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

int main(int argc, char* argv[]) {
    int v = 2015;

    char array[11];
    int i;

    int isPositive = 1;
    if (v < 0) {
        isPositive = 0;
        v = -v;
    }

    for (i = sizeof(array) - 1; v != 0; i--, v /= 10)
        array[i] = (char)((v % 10) + '0');

    if (!isPositive) {
        array[i] = '-';
        i--;
    }

    for (; i >= 0; i--)
        array[i] = ' ';

    if (write(1, array, sizeof(array)) == -1)
        perror("Print of array");
}
```

Operacje na deskryptorach

Po co są deskryptory? (1)

Deskryptor – identyfikator używany przez system operacyjny do wykonywania operacji wejścia/wyjścia na plikach.

Deskryptor reprezentowany jest przez nieujemną liczbę całkowitą.

Specjalne deskryptory:

- 0 – standard input
- 1 – standard output
- 2 – standard error

Po co są deskryptory? (2)

W systemach UNIXowych różne zasoby reprezentowane są jako pliki:

- dokumenty
- katalogi
- urządzenia zewnętrzne (dyski, karty sieciowe, klawiatury, drukarki)
- komunikacja miedzy-procesowa
- komunikacja sieciowa
- ...

Użyteczne biblioteki

```
// file control options, e.g., open(), O_CREAT
#include <fcntl.h>

// useful macros and functions
#include <stdlib.h>

// standard symbolic constants and types, e.g., STDIN_FILENO
#include <unistd.h>

// data returned by the stat() function
#include <sys/stat.h>

// datatypes, e.g., pid_t, ssize_t
#include <sys/types.h>

// common error numbers
#include <errno.h>
```

Otwieranie i tworzenie pliku

```
int creat(const char *pathname, mode_t mode);
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Parametry:

- `pathname` – nazwa pliku (w szczególności nazwa ścieżkowa)
- `flags` – tryb otwarcia:
  - `O_WRONLY`
  - `O_RDONLY`
  - `O_RDWR`
  - `O_APPEND`
  - `O_CREAT`
  - `O_TRUNC`
- `mode` – prawa dostępu (np. `S_IRUSR | S_IWUSR`, 0600)

Wartość powrotna:

- deskryptor pliku

Zamykanie i usuwanie pliku

```
int close(int fd);
int unlink(const char *pathname);
```

Parametry:

- `fd` – deskryptor pliku, który zostanie zamknięty
- `pathname` – nazwa pliku

Wartość powrotna:

- 0 w przypadku poprawnego zakończenia

Odczyt pliku

```
ssize_t read(int fd, void *buf, size_t count);
```

Parametry:

- `fd` – deskryptor pliku, z którego następuje odczyt danych
- `buf` – adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane
- `count` – liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane)

Wartość powrotna:

- liczba odczytanych bajtów (0 oznacza koniec pliku)

Odczyt pliku

```
ssize_t read(int fd, void *buf, size_t count);
```

Parametry:

- `fd` – deskryptor pliku, z którego następuje odczyt danych
- `buf` – adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane
- `count` – liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane)

Wartość powrotna:

- liczba odczytanych bajtów (0 oznacza koniec pliku)

Z deskryptorem stowarzyszony jest wskaźnik określający pozycję w pliku (tzn. ile danych zostało już przeczytanych/zapisanych).

Zapis pliku

```
ssize_t write(int fd, const void *buf, size_t count);
```

Parametry:

- `fd` – deskryptor pliku, z którego następuje odczyt danych
- `buf` – adres początku obszaru pamięci, zawierającego blok danych do zapisania
- `count` – liczba bajtów do zapisania w pliku

Wartość powrotna:

- liczba zapisanych bajtów (0 oznacza brak jakiejkolwiek modyfikacji)

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
// poprawne
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
// poprawne

int num = 89;
write(1, num, sizeof num);
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
// poprawne

int num = 89;
write(1, num, sizeof num);
// niepoprawne, &num, wypisze Y
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
// poprawne

int num = 89;
write(1, num, sizeof num);
// niepoprawne, &num, wypisze Y

int *num;
*num = 89;
write(1, num, sizeof num);
```

Poprawne czy błędne wywołania funkcji?

```
char array[10];
read(0, array, 10);
// poprawne

int num[3];
read(0, num, sizeof num);
// poprawne

int num = 89;
write(1, num, sizeof num);
// niepoprawne, &num, wypisze Y

int *num;
*num = 89;
write(1, num, sizeof num);
// niepoprawne, (int*)malloc(sizeof(int)) i sizeof *num
```

Odczyt całego pliku

```
const int MAX_COUNT = 64;
int n;
char buf[MAX_COUNT];

while ((n = read(fd, buf, MAX_COUNT)) > 0) {
    if (write(1, buf, n) == -1) {
        perror("write error");
        exit(1);
    }
}

if (n == -1) {
    perror("read error");
    exit(1);
}
```

Błędy

Szczegółowy kod błędu można odczytać badając wartość globalnej zmiennej `errno` typu `int`.

Obsługa błędów – funkcja `perror` (bada wartość zmiennej `errno` i wyświetla tekstowy opis błędu, który wystąpił).

```
#include <errno.h>
...
int fd = open("example.txt", O_RDONLY);
if (fd == -1) {
    perror("File opening");
    printf("Error code: %d\n", errno);
    exit(1);
}
```

Błędy

Szczegółowy kod błędu można odczytać badając wartość globalnej zmiennej `errno` typu `int`.

Obsługa błędów – funkcja `perror` (bada wartość zmiennej `errno` i wyświetla tekstowy opis błędu, który wystąpił).

```
#include <errno.h>
...
int fd = open("example.txt", O_RDONLY);
if (fd == -1) {
    perror("File opening");
    printf("Error code: %d\n", errno);
    exit(1);
}
```

Na slajdach obsługa błędów często jest pominięta!

Przesuwanie wskaźnika bieżącej pozycji

```
off_t lseek(int fd, off_t offset, int whence);
```

Parametry:

- `fd` – deskryptor pliku, z którego następuje odczyt danych
- `offset` – wielkość przesunięcia
- `whence` – odniesienie:
  - `SEEK_SET`
  - `SEEK_END`
  - `SEEK_CUR`

Wartość powrotna:

- aktualny offset liczony od początku pliku

Funkcje `f*()`

```
 FILE *fopen(const char *path, const char *mode);
   int fclose(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
                                     FILE *stream);
size_t fwrite(const void *ptr, size_t size,
                          size_t nmemb, FILE *stream);
   int fflush(FILE *stream);
   int fseek(FILE *stream, long offset, int whence);
  long ftell(FILE *stream);

   int fprintf(FILE *stream, const char *format, ...);
   int fscanf(FILE *stream, const char *format, ...);
 char *fgets(char *s, int size, FILE *stream);
```

Funkcje `f*()` są zdefiniowane w ANSI C,
działają na buforach, i wspierają tryb tekstowy.

Parsowanie argumentów (1)

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

int main(int argc, char *argv[]) {
    int opt;
    opterr = 0;       //disable error massages

    while ((opt = getopt(argc, argv, ":if:lr")) != -1) {
        switch (opt) {
            case 'i':
            case 'l':
            case 'r':
                printf("Option: %c\n", opt);
                break;
```

Parsowanie argumentów (2)

```
            case 'f':
                printf("Filename: %s\n", optarg);
                break;
            case ':':
                printf("Option %c needs a value\n", optopt);
                break;
            case '?':
                printf("Unknown option: %c\n", optopt);
                break;
        }
    }

    for (; optind < argc; optind++)
        printf("Argument: %s\n", argv[optind]);
    exit(0);
}
```

Procesy

Proces i jego zasoby

Proces

abstrakcyjne pojęcie określające program w trakcie wykonywania, wraz z niezbędnymi do tego wykonania zasobami systemu komputerowego

Zasoby przydzielone procesowi

- obraz kodu maszynowego stowarzyszonego z wykonywanym programem
- pamięć, która zawiera kod i dane programu, stos i stertę (stóg)
- deskryptory udostępnione przez system operacyjny
- atrybuty bezpieczeństwa (np. właściciel procesu i jego uprawnienia)
- procesor wraz z jego stanem (np. zawartość rejestrów, adresowanie w pamięci fizycznej)

Procesy w systemach UNIXowych

Proces identyfikowany jest przez unikalny process identifier (PID).

Procesy tworzą drzewiastą strukturę (patrz polecenie `pstree`).

Pierwotny proces to proces `init` (teraz `systemd`) o numerze 1.

```
$ ps -ejH
  PID  PGID   SID TTY          TIME CMD
  ...
    1     1     1 ?        00:00:10 systemd
  539   539   539 ?        00:01:47   systemd-journal
  548   548   548 ?        00:00:00   lvmetad
  562   562   562 ?        00:00:01   systemd-udevd
  913   913   913 ?        00:00:00   auditd
  930   930   930 ?        00:00:39   rsyslogd
  931   931   931 ?        00:00:59   avahi-daemon
  937   931   931 ?        00:00:00     avahi-daemon
  932   932   932 ?        00:01:07   dbus-daemon
```

Użyteczne biblioteki

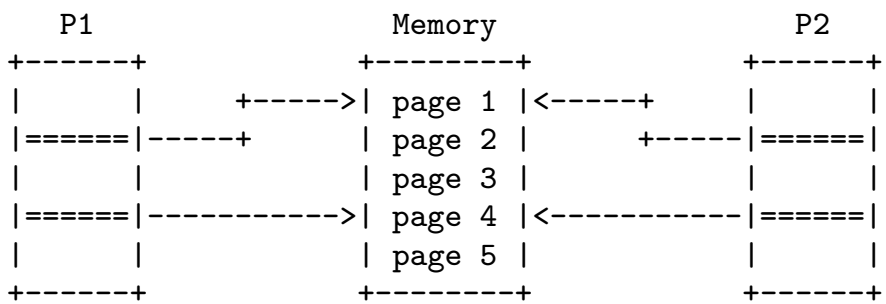
```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>

// signal related macros and functions
#include <signal.h>

// process waiting
#include <sys/wait.h>
```

Tworzenie procesu (1)

Nowy proces jest tworzony poprzez stworzenie kopii procesu rodzica za pomocą funkcji `fork()`.

Proces macierzysty i potomny mają przydzielone osobne przestrzenie pamięci wirtualnej.

Na początku oba procesy współdzielą strony w pamięci. Są one kopiowane do lokalnej przestrzeni w momencie modyfikacji – Copy-On-Write (COW).



Tworzenie procesu (2)

```
pid_t fork(void);
```

Wartość powrotna:

- 0 – w przypadku procesu-potomka
- PID procesu potomka – w przypadku procesu rodzica

Przykład:

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

int main() {
    printf("start\n");
    fork();
    printf("finish\n");
}
```

Proces macierzysty i proces potomny (1)

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

int main() {
    if (fork() == 0)
        printf("child\n");
    else
        printf("parent\n");
}
```

Identyfikator procesu i rodzica procesu:

```c
pid_t getpid(void);
pid_t getppid(void);
```

Proces macierzysty i proces potomny (2)

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

int main() {
    pid_t c = fork();

    printf("c: %10d \t pid: %10d \t ppid: %10d\n",
                                c, getpid(), getppid());

    sleep(3);
}
```

Kolejność wykonywania procesów

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

int main() {
    fork();
    printf("S\n");
    fork();
    printf("O\n");
    fork();
    printf("P\n");
}
```

Liczba stworzonych procesów (1)

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

int main() {
    fork();
    fork();
    if (fork() == 0)
        fork();
    fork();
}
```

Liczba stworzonych procesów (2)

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

int main() {
    fork();
    fork();
    if (fork() != 0)
        exit();
    fork();
}
```

Liczba stworzonych procesów (3)

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

int main() {
    fork();
    fork();
    if (fork() != 0)
        if (fork() == 0)
            fork();
        else
            exit();
    fork();
}
```

Liczba stworzonych procesów (4)

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

#define N 63

int main() {
    for (int i = 0; i < N; i++)
        if (fork() == 0) {
            printf("proces nr %d\n", i + 1);
            exit(0);
        }
}
```

Kończenie procesów (1)

```
void exit(int status);
int kill(pid_t pid, int signum);
```

Parametry:

- `status` – kod wyjścia udostępniany procesowi macierzystemu
- `pid` – identyfikator procesu, do którego adresowany jest sygnał
- `signum` – numer przesyłanego sygnału

Wartość powrotna:

- 0 w przpadku poprawnego zakończenia funkcji `kill()`

Kończenie procesów (2)

```
#include <signal.h>
#include <stdlib.h>

...

exit(7);
exit(EXIT_SUCCESS); // EXIT_FAILURE

kill(pid, SIGKILL);
```

Inf. o sposobie zakończenia potomka (1)

```
pid_t wait(int *status);
```

Funkcja zwraca indentyfikator zakończonego procesu lub -1 w przypadku błędu.

Parametry:

- `status` – adres słowa w pamięci, w którym umieszczony zostanie status zakończenia

Wartość powrotna:

- PID procesu potomnego

Inf. o sposobie zakończenia potomka (2)

Jeśli wywołanie funkcji `wait()` nastąpi przed zakończeniem procesu potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie.

Po zakończeniu procesu potomka następuje wyjście procesu macierzystego z funkcji `wait()`.

Pod adresem wskazanym w parametrze znajduje się status zakończenia.

Status zakończenia zawiera:

- numer sygnału gdy potomek zakończył działanie wskutek nieprzechwyconego sygnału
- kod wyjścia z `exit()` gdy potomek zakończył działanie poprawnie

Inf. o sposobie zakończenia potomka (3)

```
int status;
pid_t p;

if ((p = fork()) == 0) {
    printf("child sleeps...\n");
    sleep(10);
    exit(5);
} else {
    printf("child pid: %d\n", p);
    wait(&status);
    if (WIFEXITED(status)) {
        printf("%d ended with exit code %d\n",
                    p, WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("%d terminated with signal %d\n",
                    p, WTERMSIG(status));
    }
}
```

Proces sierota i proces zombie

Sierota
proces potomny, którego przodek się już zakończył
(`getppid()` zwraca 1 lub PID procesu `init`
użytkownika)

Zombie
proces potomny, który zakończył swoje działanie
i czeka na przekazanie statusu zakończenia przodkowi
(w `ps` oznaczony jako `<defunct>`)

System nie utrzymuje procesów zombie, jeśli przodek ignoruje
sygnał `SIGCLD`.

Proces sierota

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

int main() {
    pid_t c;

    if ((c = fork()) == 0)
        sleep(3);

    printf("   c = %d\n", c);
    printf(" pid = %d\n", getpid());
    printf("ppid = %d\n", getppid());
}
```

Proces zombie

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t c;

    if ((c = fork()) == 0) {
        printf("child exits\n");
        exit(0);
    } else {
        printf("child PID: %d\n", c);
        printf("parent sleeps...\n");
        sleep(15); // check ps on child's PID
        printf("parent wakes up...\n");
        if (wait(NULL) == c)
            printf("child joined\n");
    }
}
```

Funkcje `exec*()` (1)

Funkcja `fork()` służy do powielania procesu.

Funkcje `exec*()` pozwalają załadować nowy program do już działającego procesu.

`exec*()`:

- resetuje pamięć procesu
- ładuje i parsuje kod maszynowy programu wyspecyfikowanego jako argument funkcji
- przekazuje sterowanie na początek załadowanego programu

Funkcje `exec*()` nigdy nie wracają.

Typowy schemat zarządzania procesami w systemach UNIXowych: fork-and-exec.

Funkcje `exec*()` (2)

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
                                char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *file, char *const argv[],
                             char *const envp[]);
```

Parametry:

- `path` – nazwa ścieżkowa pliku z programem
- `file` – nazwa pliku z programem
- `arg` – argument linii poleceń
- `argv` – wektor (tablica) argumentów linii poleceń
- `envp` – wektor zmiennych środowiskowych

Funkcje `exec*()` (3)

```
execl("/bin/ls", "ls", "-a", NULL);

execlp("ls", "ls", "-a", NULL);

char *const av[]={"ls", "-a", NULL};
execv("/bin/ls", av);

char *const av[]={"ls", "-a", NULL};
execvp("ls", av);
```

Łącza

Łącza

Łącza są prostym mechanizmem komunikacji między procesami:

- łącza są jednokierunkowe (w POSIX)
- mają ograniczony rozmiar, zwykle 4-8 KB
- dostęp do łączy jest wyłącznie sekwencyjny
- dane odczytane z łącza są usuwane
- dane odczytywane są w kolejności, w której były zapisane (FIFO)

Operacje odczytu i zapisu są blokujące:

- `read()` na pustym łączu – jeśli jest otwarty deskryptor do tego łącza do zapisu
- `write()` – jeśli w łączu nie ma wystarczająco dużo miejsca do zapisania całego bloku

Rodzaje łącz

Potok (łącze nienazwane, ang. *pipe*)
nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza

Kolejka FIFO (łącze nazwane, ang. *named pipe*)
ma dowiązanie w systemie plików, co oznacza, że jego nazwa jest widoczna w jakimś katalogu i może ona służyć do identyfikacji łącza

Tworzenie potoku

```
int pipe(int fd[2]);
```

Parametry:

- `fd` – tablica 2 deskryptorów:
  - `fd[0]` – deskryptor potoku do odczytu
  - `fd[1]` – deskryptor potoku do zapisu

Wartość powrotna:

- 0 – w przypadku poprawnego zakończenia

Potok między dwoma procesami

```
char buf[32];
int fd[2];
int n;

if (pipe(fd) == -1) {
    perror("pipe creation error");
    exit(1);
}

if (fork() == 0) {
    close(fd[0]);
    if (write(fd[1], "Hello!\n", 7 ) == -1) {
        perror("child write error");
        exit(1);
    }
    close(fd[1]);
} else {
    close(fd[1]);
    while ((n = read(fd[0], buf, sizeof(buf))) > 0)
        write(1, buf, n);
    if (n == -1) {
        perror("parent read error");
        exit(1);
    }
    close(fd[0]);
    if (wait(NULL) == -1) {
        perror("wait on child");
        exit(1);
    }
}
```

Powielanie deskryptorów

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Parametry:

- `oldfd` – numer deskryptora, który ma zostać powielony
- `newfd` – numer nowego deskryptora, z którego można korzystać wymiennie z `oldfd`

Wartość powrotna:

- numer nowego deskryptora

W `dup()` przydzielany jest najniższy wolny numer deskryptora.

W `dup2()`, gdy `newfd` istnieje, `newfd` jest najpierw zamykany i wtedy duplikowany.

Po co powielać deskryptory? (1)

```
int fd[2];

pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    dup2(fd[1], 1);
    execlp("ls", "ls", "-l", NULL);
} else {
    close(fd[1]);
    dup2(fd[0], 0);
    execlp("tr", "tr", "a-z", "A-Z", NULL);
}
```

Po co powielać deskryptory? (2)

```
int fd[2];
int outFile;

pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    dup2(fd[1], 1);
    execlp("ls", "ls", "-l", NULL);
} else {
    close(fd[1]);
    dup2(fd[0], 0);
    outFile = open("outFile", O_CREAT|O_WRONLY|O_TRUNC,
                              S_IRUSR|S_IWUSR);
    dup2(outFile, 1);
    execlp("tr", "tr", "a-z", "A-Z", NULL);
}
```

Tworzenie kolejki FIFO

```
int mkfifo(const char *pathname, mode_t mode);
```

Parametry:

- `pathname` – nazwa pliku (w szczególności ścieżka dostępu)
- `mode` – prawa dostępu do nowo tworzonego pliku.

Wartość powrotna:

- 0 w przypadku poprawnego zakończenia

Otwieranie kolejki FIFO

```
mkfifo("myfifo", S_IRUSR | S_IWUSR);
...
fdr = open("myfifo", O_RDONLY);
fdw = open("myfifo", O_WRONLY);
```

Kolejka FIFO – przykład

```
char name[] = "myfifo";
int fd;

if (mkfifo(name, S_IRUSR | S_IWUSR) == -1) {
    perror("fifo create error");
    exit(1);
}

if (fork() == 0) {
    if ((fd = open(name, O_WRONLY)) == -1) {
        perror("child fifo open error");
        exit(1);
    }
    dup2(fd, 1);
    execlp("ls", "ls", "-l", NULL);
} else {
    if ((fd = open(name, O_RDONLY)) == -1) {
        perror("parent mkfifo error");
        exit(1);
    }
    dup2(fd, 0);
    execlp("tr", "tr", "a-z", "A-Z", NULL);
}
```

Mechanizmy IPC

Mechanizmy IPC (1)

Mechanizmy IPC (ang. *Interprocess Communication*) służą do komunikacji między procesami oraz do synchronizacji. Istnieją trzy typy zasobów:

- segmenty pamięci współdzielonej,
- tablice semaforów,
- kolejki komunikatów.

Atrybuty zasobu:

- identyfikator
- właściciel
- prawa dostępu
- publiczny klucz (opcjonalnie)

Mechanizmy IPC (2)

```
$ ipcs
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 297140224  tkob       700        3078572    2          dest
0x00000000 281378817  tkob       700        251712     2          dest
0x00000000 884738     tkob       700        4092       2          dest
0x00000000 98307      tkob       700        17400      2          dest
...

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x5402077b 131072     tkob       600        1
0x540206a8 163841     tkob       600        1
```

Przydatne komendy z linii poleceń

`ipcs` – wypisuje istniejące w systemie zasoby

`ipcrm` – usuwa podany zasób

- `ipcrm -M` – usuwa segment pamięci współdzielonej po kluczu,
- `ipcrm -m` – usuwa segment pamięci współdzielonej po id,
- `ipcrm -S` – usuwa tablicę semaforów po kluczu,
- `ipcrm -s` – usuwa tablicę semaforów po id,
- `ipcrm -Q` – usuwa kolejkę komunikatów po kluczu,
- `ipcrm -q` – usuwa kolejkę komunikatów po id.

Inne: `ipcmk`, `msgrcv`, `msgsnd`, `semget`, `semop`, `shmat`, `shmdt`, `shmget`

Interfejs programistyczny

Do każdego typu zasobu możemy wyróżnić trzy typy funkcji:

- funkcje `get` – tworzą zasób lub pobierają jego identyfikator,
- funkcje `op` – wykonują podstawowe operacje na zasobie,
- funkcje `ctl` – wykonują operacje kontrolne na zasobie.

	tworzenie (get)	operacje (op)	kontrola (ctl)
pamięć współ.	shmget	shmat / shmdt	shmctl
semafory	semget	semop	semctl
kolejki kom.	msgget	msgsnd / msgrcv	msgctl

Tworzenie klucza publicznego IPC

```
ket_t ftok(const char *pathname, int proj_id);
```

Parametry:

- `pathname` – ścieżka do istniejącego pliku (do którego jest dostęp),
- `proj_id` – identyfikator projektu (*namespace*).

Wartość powrotna:

- unikalny klucz (tworzony deterministycznie) – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Tworzenie segmentu pamięci współdzielonej

```
int shmget(key_t key, int size, int shmflg);
```

Parametry:

- `key` – klucz (dla prywatnych: `IPC_PRIVATE`),
- `size` – rozmiar obszaru pamięci współdzielonej w bajtach,
- `shmflg` – flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Wartość powrotna:

- identyfikator zasobu – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Przyłączanie segmentu pamięci współdzielonej

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

Parametry:

- `shmid` – identyfikator obszaru pamięci współdzielonej, zwrócony przez funkcję `shmget`,
- `shmaddr` – adres w przestrzeni adresowej procesu, pod którym ma być dostępny segment pamięci współdzielonej (wartość `NULL` oznacza wybór adresu przez system),
- `shmflg` – flagi, specyfikujące sposób przyłączenia (np. `SHM_RDONLY` — przyłączenie tylko do odczytu).

Wartość powrotna:

- adres segmentu (`shmat`) lub 0 (`shmdt`) – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Operacje kontrole na seg. pamięci współ.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Parametry:

- `shmid` – identyfikator obszaru pamięci współdzielonej, zwrócony przez funkcję `shmget`
- `cmd` – specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `IPC_RMID`)
- `shmid ds` – wskaźnik na strukturę opisującą atrybuty segmentu pamięci współdzielonej (np. właściciel, prawa dostępu).

Wartość powrotna:

- 0 – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Przykład 1/2

```
char *buffer;
key_t my_key;
int shmid;

if ((my_key = ftok(argv[0], 'T')) == -1) {
    perror("cannot generate public ipc key");
    exit(1);
}

if ((shmid = shmget(my_key, 100, IPC_CREAT | 0660)) == -1) {
    perror("cannot get shared memory segment");
    exit(1);
}

if ((buffer = shmat(shmid, NULL, 0)) == (void *) -1) {
    perror("cannot attach shared memory segment");
    exit(1);
}
```

Przykład 2/2

```
buffer[0] = 'a';
buffer[1] = 'b';
buffer[2] = 0;

strcpy(buffer, "tekst");

if (shmdt((const void *) buffer) == -1) {
    perror("cannot dettach shared memory segment");
    exit(1);
}

if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("cannot remove shared memory segment");
    exit(1);
}
```

Tworzenie tablicy semaforów

```
int semget(key_t key, int nsems, int semflg)
```

Parametry:

- `key` – klucz (dla prywatnych: `IPC_PRIVATE`),
- `nsems` – liczba semaforów,
- `semflg` – flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Wartość powrotna:

- identyfikator zasobu – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Wykonanie operacji semaforowej

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Parametry:

- `semid` – identyfikator zbioru semaforów, zwrócony przez funkcję `semget`,
- `sops` – adres tablicy struktur, w której każdy element opisuje operację na jednym semaforze w zbiorze wg definicji `sembuf`,
- `nsops` – rozmiar tablicy adresowanej przez `sops` (liczba elementów).

Wartość powrotna:

- 0 – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Wykonanie operacji semaforowej - c.d.

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

- `sem_num` – numer semafora, na którym ma być wykonana operacja,
- `sem_op` – wartość, która ma zostać dodana do zmiennej semaforowej,
- `sem_flg` – flagi operacji (`IPC_NOWAIT` — wykonanie bez blokowania, `SEM_UNDO` — wycofanie operacji w przypadku zakończenia procesu).

Wykonanie operacji kontrolnych na tablicy semaforów

```
int semctl(int semid, int semnum, int cmd, ...);
```

Parametry:

- `semid` – identyfikator tablicy semaforów, zwrócony przez funkcję `semget`,
- `semnum` – numer semafora,
- `cmd` – specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `SETVAL`, `GETVAL`, `SETALL`, `IPC_RMID`, itp.).

Wartość powrotna:

- wynik wykonania komendy – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Przykład

```
static struct sembuf buf;

void sem_up(int semid, int semnum) {
    buf.sem_num = semnum;
    buf.sem_op = 1;
    buf.sem_flg = 0;

    if (semop(semid, &buf, 1) == -1) {
        perror("Cannot increase semaphore value");
        exit(1);
    }
}
```

Przykład - c.d.

```
void sem_down(int semid, int semnum) {
    buf.sem_num = semnum;
    buf.sem_op = -1;
    buf.sem_flg = 0;

    if (semop(semid, &buf, 1) == -1) {
        perror("Cannot decrease semaphore value");
        exit(1);
    }
}
```

Tworzenie kolejki komunikatów

```
int msgget(key_t key, int msgflg)
```

Parametry:

- `key` – klucz (dla prywatnych: `IPC_PRIVATE`)
- `msgflg` – flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)

Wartość powrotna:

- identyfikator zasobu – w przypadku poprawnego zakończenia
- -1 – w przeciwnym przypadku

Wysyłanie komunikatu

```
int msgsnd(int msgid, const void *msgp, int msgs,
           int msgflg);
```

Parametry:

- `msgid` – identyfikator kolejki komunikatów, zwrócony przez funkcję `msgget`,
- `msgp` – wskaźnik do obszaru pamięci zawierającego komunikat,
- `msgs` – rozmiar właściwej treści komunikatu,
- `msgflg` – dodatkowe flagi (np. `IPC_NOWAIT`).

Wartość powrotna:

- 0 – w przypadku poprawnego zakończenia
- -1 – w przeciwnym przypadku

Wysyłanie komunikatu - c.d.

Ogólna postać komunikatu:

```
struct msgbuf {
    long mtype;
    char mtext[1024];
}
```

Treść komunikatu może być dowolną strukturą.

Pole `mtype` określa typ komunikatu, dzięki czemu możliwe jest przy odbiorze wybieranie z kolejki komunikatów określonego rodzaju. Typ komunikatu musi być wartością większą od 0.

Rozmiar komunikatu (`msgs`) liczymy bez pola `mtype`.

Odbieranie komunikatu

```
int msgrcv(int msgid, const void *msgp, int msgs,
           long msgtyp, int msgflg);
```

Parametry:

- `msgid` – identyfikator kolejki komunikatów, zwrócony przez funkcję `msgget`,
- `msgp` – wskaźnik do obszaru pamięci gdzie ma być zapisany komunikat,
- `msgs` – rozmiar właściwej treści komunikatu,
- `msgtyp` – typ komunikatu jaki ma być odebrany,
- `msgflg` – dodatkowe flagi (np. `IPC_NOWAIT`, `MSG_NOERROR`).

Wartość powrotna:

- liczba bajtów odczytanej wiadomości – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Odbieranie komunikatu - c.d.

`msgtyp` – typ komunikatu jaki ma być odebrany:

- > 0 wybierany jest komunikat którego typ jest dokładnie taki jak `msgtyp`,
- < 0 wybierany jest komunikat, który ma najmniejszą wartość typu mniejszą lub równą bezwzględnej wartości `msgtyp`,
- $= 0$ typ komunikatu nie jest brany pod uwagę przy wyborze.

Wykonanie operacji kontrolnych na kolejce komunikatów

```
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

Parametry:

- `msgid` – identyfikator kolejki komunikatów, zwrócony przez funkcję `msgget`,
- `cmd` – specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `IPC_RMID`, itp.),
- `msgid_ds` – wskaźnik na strukturę opisującą atrybuty kolejki komunikatów (np. właściciel, prawa dostępu).

Wartość powrotna:

- wynik wykonania komendy – w przypadku poprawnego zakończenia,
- -1 – w przeciwnym przypadku.

Wątki

Biblioteka Pthreads

Plik nagłówkowy:

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

Kompilacja:

```
$ gcc -lpthread -Wall program.c -o program
$ gcc -pthread -Wall program.c -o program
```

Tworzenie wątku (1)

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

Parametry:

- `thread` – wskaźnik na id wątku,
- `attr` – atrybuty (może być `NULL`),
- `start_routine` – funkcja obsługi wątku,
- `arg` – argument funkcji obsługi

Wartość powrotna:

- 0 – w przypadku poprawnego zakończenia,
- numer błędu – w przeciwnym przypadku.

Tworzenie wątku (2)

```
void *start_routine(void *argument) {
    printf("Hello\n");
    return NULL;
}

...

pthread_t id;
errno = pthread_create(&id, NULL, start_routine, NULL);
if (errno != 0) {
    perror("creating thread");
    exit(-1);
}
```

Tworzenie wątku (3)

```
void *start_routine(void *argument) {
    int *p = (int*) argument;
    printf("Argument = %d\n", *p);
    return NULL;
}

...

pthread_t id;
int arg = 1000;
errno = pthread_create(&id, NULL, start_routine, &arg);
if (errno != 0) {
    perror("creating thread");
    exit(-1);
}
sleep(5);
```