Systemy Operacyjne

Podstawy zarządzania procesami w systemach Windows i Linux

swernikowski@zut.edu.pl

wersja: 21-10-1

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

{lin,win} Rozpoznawanie błędów wywoływanych przez wywołania systemowe – zmienna errno

```
#include <errno.h>
extern int errno;
```

- zmienna `errno` ustawiana jest przez funkcje systemowe i podaje dokładną przyczynę ewentualnego niepowodzenia (zwykle funkcje systemowe zwracają `-1` albo `NULL` w sytuacji błędu)
- należy pamiętać, że **każda** funkcja odwołująca się do systemu może zmienić wartość `errno`, a więc poniższy schemat postępowania jest **błędny**:

```
if (somecall() == -1) {
   printf("somecall() zwróciło błąd\n");
   if (errno == ...) { ... }
}
```

- poprawne postępowanie jest następujące:

```
if (somecall() == -1) {
   int serverr = errno;
   printf("somecall() zwróciło błąd\n");
               if (serverr == ...) { ... }
}
```

{lin} Pełną listę wartości zmiennej `errno` zawiera:

http://man7.org/linux/man-pages/man3/errno.3.html

{win} Pełną listę wartości zmiennej `errno` zawiera:

https://docs.microsoft.com/pl-pl/cpp/c-runtime-library/errno-constants?view=vs-2019

{lin,win} Rozpoznawanie błędów wywoływanych przez wywołania systemowe – funkcja strerror

```
#include <string.h>
char *strerror(int errnum);
```

- funkcja zwraca treść informacji opisującej błąd o numerze podanym w argumencie
- łańcuch pod udostępnionym wskaźnikiem znajduje się statycznie wewnątrz funkcji, więc **nie wolno** go zmieniać, a kolejne wywołanie funkcji z dużym prawdopodobieństwem go zmodyfikuje

{lin,win} Rozpoznawanie błędów wywoływanych przez wywołania systemowe:

```
// error.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main(void) {
    void *p = malloc(-1); // jak wygląda -1 w zapisie U2?
    if(p == NULL) {
        int e = errno;
        printf("errno=%d: '%s'\n", e, strerror(e));
    }
    return 0;
}
```

Standard języka C dopuszcza **trzy** postaci nagłówka funkcji `main()`:

1) bez parametrów:

```
int main(void)
```

2) z dwoma parametrami:

```
int main(int argc, char **argv)
```

3) z trzema parametrami:

```
int main(int argc, char **argv, char **env)
```

```
int main(int argc, char **argv)
```

`argc`
liczba argumentów przekazanych w linii poleceń + 1

`argv`
argumenty (łańcuchy znaków) przekazane do programu, przy czym:

`argv[0]` - nazwa programu
`argv[1]` - pierwszy argument
`:`
`argv[argc-1]` - ostatni argument
`argv[argc]` - NULL

{lin,win} Wypisanie na `stdout` wszystkich argumentów programu:

```
// args.c
#include <stdio.h>

int main(int argc, char **argv)
{
    for(int i = 0; i < argc; i++)
        printf("%2d: %s\n", i, argv[i]);
    return 0;
}
```

{lin,win} Ten sam efekt, ale nieco inaczej:

```
// args_2.c
#include <stdio.h>

int main(int argc, char **argv)
{
    int i = 1;
    char **p = argv;
    while(*p)
        printf("%2d: %s\n", i++, *p++);
    return 0;
}
```

{win} Zakładając, że plik wykonywalny nazywa się `prog.exe`:

```
C:\>prog
 0: prog

C:\>prog ala ma kota
 0: prog
 1: ala
 2: ma
 3: kota

C:\>dir /w
prog.c       prog.obj         prog.exe

C:\>prog *
 0: prog
 1: *
```

{lin} Zakładając, że plik wykonywalny nazywa się `prog`:

```
$ ./prog
 0: prog

$ ./prog ala ma kota
 0: ./prog
 1: ala
 2: ma
 3: kota

$ ls
prog.c      prog

$ ./prog *
 0: ./prog
 1: prog.c
 2: prog
```

```
int main(int argc, char **argv, char **env)
```

`env`
jak `argv`, ale jest tablicą wskaźników na łańcuchy tworzące tzw. **zmienne środowiska** (ang. *environment variables*); w ostatnim elemencie tablicy znajduje się `NULL`

Zmienne środowiska:

- łańcuchy postaci "NAME=VALUE", przechowywane wewnątrz obszaru pamięci każdego procesu
- każdy proces ma własną kopię pełnego zestawu zmiennych
- proces może modyfikować **własny** zestaw zmiennych (usuwać zmienne, dodawać zmienne, zmieniać ich wartość), ale nie ma dostępu do zmiennych innych procesów
- nowy proces (zwykle) otrzymuje kopię środowiska swojego rodzica, ale można to zmienić

{lin,win} Wypisanie na `stdout` wszystkich dostępnych zmiennych środowiska:

```
// env.c
#include <stdio.h>

int main(int argc, char *argv[], char **env) {
    char **p = env;
    int    n = 1;
    while(*p)
        printf("%2d: %s\n", n++, *p++);
    return 0;
}
```

{lin,win} Funkcja `getenv()`: pobranie łańcucha wskazanej zmiennej środowiska:

```
#include <stdlib.h>
char *getenv(char *name);
```

Parametry:

- `name` – nazwa żądanej zmiennej

Wynik:

- `NULL` jeśli w środowisku nie ma zmiennej o podanej nazwie
- w przeciwnym przypadku wskaźnik na łańcuch przechowujący zmienną

{lin,win} Wypisanie na `stdout` wartości zmiennej PATH
(zmienna PATH istnieje i w Linuksie, i w Windows, chociaż z inną składnią):

```
// path.c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *nam = "PATH";
    char *val = getenv(nam);
    if(val == NULL)
        printf("%s undefined\n", nam);
    else
        printf("%s=%s\n", nam, val);
    return 0;
}
```

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

{lin} Funkcja `setenv()`: utworzenie nowej zmiennej bądź modyfikacji zmiennej już istniejącej:

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

Parametry:

- `name` – nazwa zmiennej
- `value` – wartość zmiennej
- `overwrite` – jeśli różne od zera, wyraża zgodę na **nadpisanie** istniejącej wartości zmiennej

Wynik:

- `0` – powodzenie
- `-1` – niepowodzenie (`errno` określa przyczynę)

Uwaga: jeśli `overwrite==0` i zmienna `name` już istnieje, funkcja zwraca 0.

{lin} Nadanie/zmiana wartości zmiennej:

```
// lin_setenv.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *nam = "VAR";
    char *val = getenv(nam);
    printf("%s\n", val ? val : "(null)");
    setenv(nam,"VAL",0);
    printf("%s\n", getenv(nam));
    return 0;
}
```

{lin} Funkcja `unsetenv()` : usunięcie zmiennej:

```
#include <stdlib.h>
int unsetenv(const char *name);
```

Parametry:

- `name` – nazwa zmiennej

Wynik:

- `0` – powodzenie
- `-1` – niepowodzenie (`errno` określa przyczynę)

{lin} Nadanie/zmiana/usunięcie wartości zmiennej:

```
// lin_unsetenv.c
#include <stdio.h>
#include <stdlib.h>

void p(char *s) {
    printf("%s\n", s ? s : "(null)");
}

int main(void)
{
    char *nam = "VAR";
    p(getenv(nam));
    setenv(nam,"VAL",0);
    p(getenv(nam));
    unsetenv(nam);
    p(getenv(nam));
    return 0;
}
```

{lin,win} Funkcja `putenv()`: utworzenie, modyfikacja lub usunięcie zmiennej:

```
#include <stdlib.h>
int putenv(const char *envstring);
```

Parametry:

- `envstring` – łańcuch określający czynność do wykonania:

  `"NAME=VALUE"` – nadanie zmiennej `NAME` wartości `VALUE`
  `"NAME"` – usunięcie zmiennej `NAME`

Wynik:

- `0` – powodzenie
- `-1` – niepowodzenie (`errno` określa przyczynę)

{lin,win} Nadanie/zmiana wartości zmiennej:

```
// putenv.c
#include <stdio.h>
#include <stdlib.h>

void p(char *s) {
    printf("%s\n", s ? s : "(null)");
}
int main(void)
{
    char *nam = "VAR", envstr[20];
    p(getenv(nam));
    sprintf(envstr,"%s=%s",nam,"VAL");
    putenv(envstr);
    p(getenv(nam));
    sprintf(envstr,"%s=",nam);
    putenv(envstr);
    p(getenv(nam));
    return 0;
}
```

{lin,win} Proces kończy swoje działanie na skutek:

- wykonania instrukcji

  ```
  return status;
  ```

  wewnątrz funkcji `main()`

- wywołania funkcji

  ```
  exit(status);
  ```

{lin,win} Funkcja `exit()` : zakończenie procesu:

```
#include <stdlib.h>
void exit(int status);
```

Parametry:

- `status` – kod zakończenia procesu (`EXIT_SUCCESS`, `EXIT_FAILURE`, generalnie `0` to zakończenie poprawne, inna wartość - niepoprawne)

Wynik:

- **brak** – z tej funkcji nie ma powrotu

{lin,win} Funkcja `system()`: wykonanie polecenia powłoki

```
#include <stdlib.h>
int system(const char *command);
```

Parametry:

- `command` – `NULL` albo polecenie powłoki wraz z ewentualnymi argumentami; składnia polecenia musi być zgodna z konwencjami systemu operacyjnego; argument o wartości `NULL` używany jest do sprawdzenia dostępności powłoki

Wynik:

- jeśli `command == NULL`:
  - `=0` – powłoka nie jest dostępna
  - `≠0` – powłoka jest dostępna
- jeśli `command !=NULL`:
  - `-1` – wykonanie polecenia zakończyło się niepowodzeniem
  - `≥0` – kod powrotu zwrócony przez powłokę

Interpretacja wyniku funkcji `system()` zależy od systemu.

{win}
Wynik jest wprost kodem powrotu z powłoki albo uruchomionego przez nią programu.

{lin}
- 8 starszych bitów wyniku jest kodem powrotu z powłoki albo uruchomionego przez nią programu
- 8 młodszych bitów jest tzw. *termination reason code*
- ustalenie faktycznego kodu powrotu umożliwia makro `WEXITSTATUS()`:

```
int ret = system(cmd);
int aret = WEXITSTATUS(ret);
```

{lin,win} Uruchomienie polecenia „`dir`” za pośrednictwem powłoki

```
// system.c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int v = system(NULL);
    printf("Shell is %s available\n", v ? "" : "not"); // ...is
available
    if(v) {
        v = system("dir");
        printf("Shell returned %d\n", v); // 0
    }
    return 0;
}
```

{lin} Uruchomienie polecenia „`false`” za pośrednictwem powłoki

```
// lin_system.c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int v = system(NULL);
    printf("Shell is %s available\n", v ? "" : "not");  // ...is
available
    if(v) {
        v = system("false");
        printf("Shell returned %d\n", v); // 256
    }
    return 0;
}
```

{lin} Funkcja `fork()`: utworzenie nowego procesu

```
#include <unistd.h>
pid_t fork(void);
```

Działanie:
- zatrzymanie procesu, który wywołał `fork()`
- sklonowanie procesu w efekcie czego powstaje nowy proces potomny (dziecko), który jest idealną kopią procesu macierzystego (rodzica)
- wznowienie obu procesów (rodzica i dziecka)
- od tego momentu oba procesy pracują niezależnie i równolegle

Wynik:
- `-1` – błąd: proces potomny nie został utworzony
- `=0` – proces potomny został utworzony, a proces, który otrzymał tę wartość, jest **dzieckiem**
- `>0` – proces potomny został utworzony, proces, który otrzymał tę wartość jest **rodzicem**, a sama wartość jest **identyfikatorem** dziecka

{lin} Funkcja `fork()`: co to jest „*fork bomb*”?

```
# include <unistd.h>

int main(void) {
 for(;;)
    fork();
}
```

{lin} Funkcje `getpid()`: pobranie identyfikatora procesu bieżącego

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

Wynik:

- identyfikator procesu, który wywołał funkcję (wywołanie **zawsze** kończy się sukcesem)

{lin} Funkcje `getppid()`: pobranie identyfikatora procesu macierzystego

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid(void);
```

Wynik:

- identyfikator procesu, który utworzył proces wywołujący funkcję (wywołanie zawsze kończy się sukcesem)

{lin} Funkcja `sleep()`: zawieszenie procesu wywołującego

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Działanie:
- zawieszenie procesu, który wywołał `sleep()`, na **co najmniej** tyle sekund, ile podano w argumencie

Parametr:
- `seconds` – liczba sekund, na które należy uśpić proces

Wynik:
- `0` jeśli cały żądany czas upłynął

{lin} Rozdwojenie procesu:

```
// lin_fork.c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
int main(void) {
    int v = fork();
    if(v == 0) {        /* child */
        printf("I'm a child %d of parent %d\n",getpid(),getppid());
        return 0;
    } else if(v > 0) {  /* parent */
        printf("I'm a parent %d of child %d\n",getpid(),v);
        return 0;
    } else {        /* failure */
        printf("fork() failed - %s\n", strerror(errno));
        return 1;
    }
}
```

{lin} Funkcja `fork()`:

- w trakcie wykonania funkcji `fork()` nowy proces otrzymuje od rodzica kopię jego danych
- kod procesu najczęściej nie podlega kopiowaniu, ponieważ z reguły jest przechowywany w pamięci „*execute only*”, której zawartość nie może być modyfikowana – tym samym dowolna liczba procesów potomnych może posługiwać się tym samym kodem
- obszary danych dziecka i rodzica są **rozłączne**

{lin} Rozłączność danych rodzica i dziecka :

```
// lin_fork_data.c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
char text[] = "PARENT";
int main(void) {
    int v = fork();
    if(v == 0) {           /* child */
        strcpy(text, "CHILD");
        printf("I'm a child: %s\n", text);
        return 0;
    } else if(v > 0) {     /* parent */
        sleep(1);
        printf("I'm a parent: %s\n", text);
        return 0;
    } else {      /* failure */
        printf("fork() failed - %s\n", strerror(errno));
        return 1;
    }
}
```

{lin} Funkcja `wait()`:

- rodzic może poczekać, aż proces dziecka zostanie zakończony i wstrzymać swoje działanie do tej chwili
- normalnie rodzic wykonuje się niezależnie od dziecka i jest niewrażliwy na fakt jego zakończenia

{lin} Funkcja `wait()`: odczekanie do zakończenia dowolnego procesu potomnego

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

Działanie:

- zawieszenie procesu, który wywołał `wait()`, do chwili zakończenia dowolnego z jego procesów potomnych; jeśli w chwili wywołania nie istnieje żaden „żywy” proces potomny, powrót z wait() następuje natychmiast

Parametr:

- `wstatus` – `NULL` albo wskaźnik na daną, w której `wait()` umieści kod zakończenia dziecka

Wynik:

- `-1` w przypadku błędu, w tym, jeśli nie ma żadnych procesów potomnych albo PID zakończonego dziecka

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

{lin} Funkcja `wait()`: odczekanie do zakończenia dowolnego procesu potomnego

```
// lin_wait.c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void) {
     int pid = fork();
     if(pid == 0) {
          printf("child with PID=%d\n",getpid());
          sleep(5);
          printf("child wakes up\n");
          return 27;
     } else if(pid == -1) {
          fprintf(stderr, "Fork failed: %s\n", strerror(errno));
          return 1;
     }
     printf("parent is waiting for child to finish\n");
     int status, child = wait(&status);
     printf("child finished: PID=%d, retcode=%d\n", child, WEXITSTATUS(status));
     return 0;
}
```

{lin} Funkcja `waitpid()`: odczekanie do zakończenia wskazanego procesu potomnego

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Działanie:

- zawieszenie procesu, który wywołał `waitpid()`, do chwili zakończenia procesu potomnego o wskazanym identyfikatorze; jeśli w chwili wywołania taki proces potomny nie istnieje, powrót z `waitpid()` następuje natychmiast

Parametr:

- `pid` – id procesu potomnego albo `-1` jeśli czeka się na dowolny z procesów potomnych
- `wstatus` – jak w przypadku `wait()`
- `options` – `0` dla zachowania domyślnego, `WNOHANG` jeśli `waitpid()` ma wrócić natychmiast, jeśli wszystkie dzieci są aktywne

Wynik:

- `-1` w przypadku błędu, w tym, jeśli nie ma żadnych procesów potomnych albo PID zakończonego dziecka

{lin} Funkcja `waitpid()`: odczekanie do zakończenia wskazanego procesu potomnego

```
// lin_waitpid.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void) {
     int pids[2];
     for(int c = 0; c < 2; c++) {
               int pid = fork();
               if(pid == 0) {
                    printf("Child #%d PID=%d\n",c,getpid());
                    sleep(1 + c);
                    return 1 + c;
               }
               pids[c] = pid;
     }
     for(int c = 1; c >= 0; c--) {
          int status, child = waitpid(pids[c],&status,0);
          printf("Parent: child #%d exited: PID=%d, retcode=%d\n", c, child, WEXITSTATUS(status));
     }
     return 0;
}
```

{lin} Rodzina funkcji `execxx()` : wymiana kodu procesu na pobrany z pliku

- funkcje z rodziny `execxx()` służą do załadowania z pliku nowego kodu i uruchomienie go w miejsce obecnie wykonywanego
- oznacza to, że funkcje tej rodziny nie wracają do miejsca wywołania, ponieważ z chwilą ich pomyślnego wykonania miejsce wywołania przestaje istnieć
- PID procesu, który wymienia kod, pozostaje bez zmian
- w przypadku błędu wszystkie funkcje tej rodziny zwracają `-1`
- każda funkcji jest odpowiedzialna za przygotowanie kompletnej tablicy `argv` uruchomianego procesu (włącznie z `argv[0]`)
- w zależności od funkcji robi się to albo podając kolejne argumenty wywołania, albo wskazując tablicę łańcuchów (zakończoną przez `NULL`), która zostanie przekopiowana do uruchamianego programu

{lin} Rodzina funkcji `execxx()` : wymiana kodu procesu na pobrany z pliku

```
#include <unistd.h>
int execl(char *path, char *arg, ... /* NULL */);
int execle(char *path, char *arg, .../* NULL, char *envp[] */);
int execlp(char *file, char arg, ... /* NULL */);
int execv(char *path, char *argv[]);
int execve(char *file, char *argv[], char *envp[]);
int execvp(char *file, char *argv[]);
int execvpe(char *file, char *argv[], char *envp[]);
```

Konwencje nazewnicze

- przyrostek **l** oznacza, że funkcja używa zmiennej listy argumentów do przekazania parametrów do uruchamianego programu (`NULL` kończy listę)
- przyrostek **p** oznacza, że funkcja używa zmiennej środowiska PATH do znalezienia pliku wykonalnego; funkcje bez tego przyrostka muszą podać pełną ścieżkę do pliku
- przyrostek **e** oznacza, że funkcja preparuje nowe środowisko do uruchamianego programu; funkcje bez tego przyrostka kopiują istniejące środowisko
- przyrostek **v** oznacza, że funkcja używa tablicy do przekazania argumentów do programu

{lin} Program doświadczalny – usypia na podaną liczbę sekund i odlicza czas

(skompilowany do pliku wykonalnego `lin_prog`)

```
// lin_prog.c -> gcc lin_prog.c -o lin_prog
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
     if(argc != 2) {
          fprintf(stderr, "usage: %s seconds\n", argv[0]);
          return 1;
     }
     char *endptr;
     int   secs = strtol(argv[1], &endptr, 10);
     if(secs <= 0 || *endptr != '\0') {
          fprintf(stderr, "argument must be an int greater than 0 - exiting\n");
          return 2;
     }
     printf("#%d will wait %d seconds...\n", getpid(), secs);
     for(int i = 0; i < secs; i++) {
          printf("#%d -> %d...\n", getpid(), secs - i);
          sleep(1);
     }
     printf("#%d completed\n", getpid());
     return 0;
}
```

{lin} Program doświadczalny – przykładowe uruchomienie:

```
$ ./lin_prog 5
#23840 will wait 5 seconds...
#23840 -> 5...
#23840 -> 4...
#23840 -> 3...
#23840 -> 2...
#23840 -> 1...
#23840 completed

$
```

{lin} Funkcja `execlp()` – argumenty wywołania programu przekazane w kolejnych argumentach funkcji

```
// lin_execlp.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>
int main(void) {
      for(int c = 0; c < 3; c++) {
            int pid = fork();
            if(pid == 0) {
                  srand(getpid());
                  int secs = rand() % 5 + 1;
                  char buf[10];
                  sprintf(buf, "%d", secs);
                  execlp("./lin_prog", "./lin_prog", buf, NULL);
                  fprintf(stderr, "exec failed: %s\n", strerror(errno));
                  return 1;
            }
      }
      for(int c = 0; c < 3; c++) {
            int status, child = wait(&status);
            printf("Parent: child #%d exited: PID=%d, retcode=%d\n", c, child, WEXITSTATUS(status));
      }
      return 0;
}
```

{lin} Przykładowe uruchomienie:

```
$ ./lin_execlp 5
#28688 will wait 1 seconds...
#28687 will wait 1 seconds...
#28688 -> 1...
#28687 -> 1...
#28689 will wait 1 seconds...
#28689 -> 1...
#28687 completed
#28688 completed
Parent: child #0 exited: PID=28688, retcode=0
Parent: child #1 exited: PID=28687, retcode=0
#28689 completed
Parent: child #2 exited: PID=28689, retcode=0

$
```

{lin} Funkcja `execvp()` – argumenty wywołania programu w tablicy

```
// lin_execvp.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void) {
      for(int c = 0; c < 3; c++) {
            int pid = fork();
            if(pid == 0) {
                  srand(getpid());
                  int secs = rand() % 3 + 1;
                  char buf[10];
                  sprintf(buf, "%d", secs);
                  char *newargv[3] = { "./lin_prog",  buf, NULL };
                  execvp("./lin_prog", newargv);
                  fprintf(stderr, "exec failed: %s", strerror(errno));
                  return 1;
            }
      }
      for(int c = 0; c < 3; c++) {
            int status, child = wait(&status);
            printf("Parent: child #%d exited: PID=%d, retcode=%d\n", c, child, WEXITSTATUS(status));
      }
      return 0;
}
```

{win}

- funkcje systemu służące tworzeniu i obsłudze procesów mocno uwidaczniają różnice w założeniach leżących u podstaw Linuksa i Windows
- Linux preferuje małe, lekkie funkcje, wykonujące ściśle określone, wąskie czynności
- Windows preferuje duże, bardzo uniwersalne funkcje o szerokich, parametryzowanych możliwościach
- np. Windows nie zna funkcji `fork()` ani nie posiada jej bezpośredniego odpowiednika, chociaż podobna funkcjonalność jest schowana we wnętrzu innych funkcji

{win} Rodzina funkcji `execxx()`: wymiana kodu procesu na pobrany z pliku

```
#include <process.h>

intptr_t execl(char *cmdname, char *arg0, ... const char *argn, NULL);
intptr_t execle(char *cmdname, char *arg0, ... const char *argn, NULL, char
**envp);
intptr_t execlp(char *cmdname, char *arg0, ... const char *argn, NULL);
intptr_t execlpe(char *cmdname, char *arg0, ... char *argn, NULL, char
**envp);
intptr_t execv(char *cmdname, char *const *argv);
intptr_t execve(char *cmdname, char *const *argv, char *const *envp);
intptr_t execvp(char *cmdname, char *const *argv);
intptr_t execvpe(char *cmdname, char *const *argv, char **envp);
```

- znaczenie przyrostków jest takie same jak w wersji linuksowej
- w razie poprawnego wykonania funkcja nie wraca do miejsca wywołania
- w razie błędu funkcja zwraca -1, a d[illegible] przez `errno`

{win} Program doświadczalny – wyprowadza na `stdout` zawartość `argv` i wartość zmiennej środowiska o nazwie `Path` (skompilowany do pliku wykonalnego `win_prog.exe`)

```
// win_prog.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[], char **envp) {
     for(int i = 0; i < argc; i++)
          printf("\nargv[%d] = '%s'", i, argv[i]);
     printf("\n\n");
     char **p = envp;
     char *path = "Path=";
     int   len  = strlen(path);
     while(*p != NULL) {
          if(strncmp(path, *p, len) == 0) {
               printf("%s\n", *p);
               break;
          }
          p++;
     }
     return 0;
}
```

{win} Przykładowe uruchomienie:

```
C:\> prog.exe ala ma kota
argv[0] = 'prog.exe'
argv[1] = 'ala'
argv[2] = 'ma'
argv[3] = 'kota'

Path=C:\BIN

C:\>
```

{win} Funkcja `execle()` – argumenty wywołania programu w argumentach funkcji + nowe środowisko

```
// win_execle.c
#include <process.h>
#include <stdio.h>

int main(int argc, char *argv[], char **envp) {
   char *myenv[2] = {
      "Path=This Is The Path I Like.",
       NULL
   };
   execle("win_prog.exe", "win_prog.exe", "arg1", "arg2", NULL,
myenv);
    printf("Something really bad happened - we should not be here!\
n");
    return 1;
}
```

{win} Funkcja `execve()` – argumenty wywołania programu w tablicy + oryginalne środowisko

```
// win_execve.c
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

int main(int argc, char *argv[], char **envp) {
    char *myargv[5];
    myargv[0] = "win_prog.exe";
    for(int i = 1; i < 4; i++) {
        myargv[i] = malloc(2);
        myargv[i][0] = '0' + i;
        myargv[i][1] = '\0';
    }
    myargv[4] = NULL;
    execve("win_prog.exe", myargv, envp);
    printf("Something really bad happened - we should not be here!\n");
    return 1;
}
```

{win} Rodzina funkcji `spawnxx()`: utworzenie nowego procesu z kodem pobranym z pliku

```
#include <process.h>

intptr_t spawnl(int mode, char *cmdname, char *arg0, ... const char *argn, NULL);
intptr_t spawnle(int mode, char *cmdname, char *arg0, ... const char *argn, NULL,
char **envp);
intptr_t spawnlp(int mode, char *cmdname, char *arg0, ... const char *argn,
NULL);
intptr_t spawnlpe(int mode, char *cmdname, char *arg0, ... char *argn, NULL, char
**envp);
intptr_t spawnv(int mode, char *cmdname, char *const *argv);
intptr_t spawnve(int mode, char *cmdname, char *const *argv, char *const *envp);
intptr_t spawnvp(int mode, char *cmdname, char *const *argv);
intptr_t spawnvpe(int mode, char *cmdname, char *const *argv, char **envp);
```

- w porównaniu do `execxx()` funkcje te mają dodatkowy pierwszy parametr, określający sposób wystartowania nowego procesu
- w razie błędu funkcja zwraca `-1`, ale w zależności od `mode` możliwe są również inne wyniki (to też odróżnia te funkcje od `execxx()`)

{win} Rodzina funkcji `spawnxx()`: utworzenie nowego procesu z kodem pobranym z pliku

Możliwe wartość parametru `mode`:

- `_P_OVERLAY` – wymień kod bieżącego procesu i wykonaj go (innymi słowy, działaj jak `exec()`)
- `_P_WAIT` – czekaj na zakończenie nowego procesu (tryb synchroniczny/blokujący)
- `_P_NOWAIT` – wykonaj kod nowego procesu równolegle z bieżącym (tryb asynchroniczny/nieblokujący)
- `_P_DETACH` – wykonaj nowy proces w tle (odłącz go od strumieni `stdin`, `stderr` i `stdout`)

{win} Rodzina funkcji `spawnxx()`: utworzenie nowego procesu z kodem pobranym z pliku

Wyniki zwracane przez funkcje zależą od wartości parametru `mode`:

- `_P_OVERLAY` – znaczenie ma tylko wartość -1 oznaczająca błąd (errno precyzuje błąd)
- `_P_WAIT` – `-1` w przypadku błędu albo kod zakończenie nowego procesu
- `_P_NOWAIT`, `_P_DETACH` – `-1` w przypadku błędu albo tzw. **uchwyt** (ang. *handle*) wystartowanego nowego procesu (uwaga – to nie jest PID!)

{win} Funkcja `spawnve()` – argumenty wywołania programu w tablicy + nowe środowisko

```
// win_spawnve.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>
#include <errno.h>

int main(void) {
     char *myenvp[] = {"Path=PATHx", NULL };
     char *myargv[] = {"win_prog.exe", "processx", NULL };
     for(int i = 0; i < 3; i++) {
          myargv[1][strlen(myargv[1]) - 1] = '1' + i;
          myenvp[0][strlen(myenvp[0]) - 1] = '1' + i;
          int ret = spawnve(_P_NOWAIT, "win_prog.exe", myargv, myenvp);
          printf("Spawn #%d: retcode=%d", i+1, ret);
          if(ret < 0) printf(" errno=%d", errno);
          printf("\n");
     }
     return 0;
}
```

{win} Natywny interfejs zarządzania procesami

```
#include <windows.h>

typedef struct _STARTUPINFO { … }
STARTUPINFO, *LPSTARTUPINFO;
```

- natywny interfejs zarządzania procesami w Windows jest (w porównaniu z linuksowym) ciężki i złożony
- jego omawianie rozpoczniemy od omówienia struktury `STARTUPINFO`
- struktura ma 18 pól – większość z nich służy do opisania wyglądu okna tworzonego procesu itp.
- pierwsze z pól o nazwie `cb` opisuje faktyczny rozmiar struktury, aby użytkownik mógł użyć tylko wybranych pól
- pole o nazwie `dwFlags` jest maską bitową informującą system, które z pól są faktycznie użyte
- pełny opis struktury jest dostępny tutaj: https://docs.microsoft.com/pl-pl/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa
- w przypadku aplikacji konsolowej (jak nasza) struktura ta wykorzystywana jest do wykonania ewentualnych przekierowań strumieni `stdin`, `stdout` i `stderr`

{win} Natywny interfejs zarządzania procesami

```
#include <windows.h>

typedef struct _PROCESS_INFORMATION { … }
PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

- kolejna używana w interfejsie struktura to `PROCESS_INFORMATION`
- jej dwa pola są interesujące dla naszych rozważań:
  - `HANDLE hProcess` – uchwyt nowego procesu (nie PID)
  - `DWORD dwProcessId` – PID nowego procesu
- uwaga:
  - `HANDLE` jest synonimem typu `void*`
  - `DWORD` jest synonimem 32-bitowego typu `unsigned int`
- pełny opis struktury jest dostępny tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/api/processthreadsapi/ns-processthreadsapi-process_information
- opis natywnych typów Windows jest dostępny tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/winprog/windows-data-types

{win} Natywny interfejs zarządzania procesami

```
#include <windows.h>

DWORD GetLastError(void);
```

- do pobierania kodu zakończenia ostatnio wywołanej usługi systemowej służy funkcja `GetLastError()`
- pełny opis funkcji jest dostępny tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror
- pełna lista kodów błędów jest dostępna tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/debug/system-error-codes

{win} Natywny interfejs zarządzania procesami

```
#include <windows.h>

DWORD FormatMessage(
  DWORD   dwFlags,
  LPCVOID lpSource,
  DWORD   dwMessageId,
  DWORD   dwLanguageId,
  LPTSTR  lpBuffer,
  DWORD   nSize,
  va_list *Arguments
);
```

- do skonstruowania czytelnego tekstu z informacją o błędzie służy funkcja `FormatMessage()`
- pełny opis funkcji jest dostępny tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/api/winbase/nf-winbase-formatmessage

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `FormatMessage()`:

`dwFlags` - w naszym przypadku zawsze `FORMAT_MESSAGE_FROM_SYSTEM`
`lpSource` - wskaźnik na wstępnie sformatowany tekst lub `NULL`
`dwMessageId` - kod błędu
`dwLanguageId` - identyfikator języka - w naszym przypadku `MAKELANGID(LANG_NEUTRAL,SUBLANG_DEFAULT)`
`lpBuffer` - wskaźnik na bufor, w którym funkcja umieści sformatowany tekst
`nSize` - rozmiar bufora
`Arguments` - lista argumentów do wypełnienia tekstu dostępnego pod adresem `lpSource`

{win} Natywny interfejs zarządzania procesami

```
#include <windows.h>

int CreateProcessA(
char                *lpApplicationName,
char                *lpCommandLine,
SECURITY_ATTRIBUTES*lpProcessAttributes,
SECURITY_ATTRIBUTES    *lpThreadAttributes,
int                    bInheritHandles,
int                    dwCreationFlags,
void                *lpEnvironment,
char                *lpCurrentDirectory,
STARTUPINFO            *lpStartupInfo,
PPROCESS_INFORMATION   *lpProcessInformation
);
```

- nowy proces jest tworzony i startowany przez funkcję `CreateProcessA()`
- pełny opis funkcji dostępny jest tutaj:
  https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

`char *lpApplicationName`

- nazwa programu do uruchomienia jako kod nowego procesu
- funkcja nie używa zmiennej `PATH` do odnalezienia pliku wykonalnego
- jeśli nazwa nie określa katalogu, zakłada się katalog bieżący
- jeśli argument jest ustawiony na `NULL`, nazwy programu szuka się wewnątrz argumentu `lpCommandLine`

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

```
char   *lpCommandLine
```

- łańcuch zawierający argumenty dla uruchamianego programu (tak jakby zostały wpisane do konsoli)
- może być ustawiony na `NULL`, jeśli program nie wymaga argumentów

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

```
SECURITY_ATTRIBUTES      *lpProcessAttributes
SECURITY_ATTRIBUTES     *lpThreadAttributes
```

- tzw. deskryptor zabezpieczeń (ang. *security descriptor*) dla nowego programu i jego głównego wątku
- jeśli są ustawione na `NULL`, używa się deskryptorów domyślnych (co skrzętnie wykorzystamy)

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

`int bInheritHandles`

- jeśli ustawiony jest na wartość różną od zera, nowy proces odziedziczy wszystkie uchwyty utworzone przez rodzica (w Windows temin „*handle*” odnosi się do wszelkich zasobów tworzonych przez proces, a zarządzanych przez OS, w tym do otwartych plików)
- ustawienie parametru na `0` wyłącza dziedziczenie

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

`int dwCreationFlags`

- ustawia priorytet dla nowego procesu (w Windows wyróżnia się 6 klas priorytetów: `ABOVE_NORMAL_PRIORITY_CLASS`, `BELOW_NORMAL_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `IDLE_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS`, `REALTIME_PRIORITY_CLASS`)
- ustawienie argumentu na 0 jest równoznaczne z `NORMAL_PRIORITY_CLASS`

Fundusze Europejskie Wiedza Edukacja Rozwój | Rzeczpospolita Polska | Unia Europejska Europejski Fundusz Społeczny

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

`void *lpEnvironment`

- wskaźnik na środowisko, które otrzyma nowy proces albo `NULL` jeśli nowy proces ma odziedziczyć środowisko rodzica
- środowisko dla funkcji `CreateProcessA()` przygotowuje się inaczej niż to, które otrzymuje funkcja `main()`
- jest to łańcuch ze zmiennymi i ich wartościami rozdzielonymi znakami *null* (`'\0'`) i zakończony znakiem *null*
- oznacza to, że środowisko widziane przez funkcję `main()` jako:

```
char *env[] = { "VAR1=1", "VAR2=2", NULL };
```

należy przygotować w postaci:

```
"VAR1=1\0VAR2=2\0"
```

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

```
char *lpCurrentDirectory
```

- łańcuch określający katalog bieżący dla nowego procesu albo NULL, jeśli proces ma pracować w tym samym katalogu bieżącym co rodzic

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

```
STARTUPINFO   *lpStartupInfo
```

- efektywna konfiguracja nowego procesu
- praktycznie nieużyteczne w przypadku aplikacji konsolowych (być może z wyjątkiem pól określających przekierowanie strumieni `stdin`, `stderr` i `stdout`)

{win} Natywny interfejs zarządzania procesami

Parametry funkcji `CreateProcessA()`:

```
PROCESS_INFORMATION  *lpProcessInformation
```

- efektywna konfiguracja uruchomionego nowego procesu
- uchwyty określone wewnątrz struktury muszą być jawnie zamknięte po zakończenie procesu potomnego

{win} Natywny interfejs zarządzania procesami

Funkcja `WaitForSingleObject()`:

```
int WaitForSingleObject(
     HANDLE hHandle,
     DWORD  dwMilliseconds
);
```

- rodzic może poczekać na zakończenie jednego z procesów potomnych, używając funkcji `WaitForSingleObject()`
- uwaga: funkcję zaprojektowano dla wielu różnych zastosowań (nie tylko do czekania na zakończenie procesów)
- argumenty określają:
  - `hHandle` – uchwyt procesu potomnego
  - `dwMilliseconds` – maksymalny czas oczekiwania w milisekundach albo INFINITE jeśli decydujemy się czekać do skutku
  - w przypadku procesów funkcja zwraca:
    - `WAIT_OBJECT_0` – proces potomny się zakończył
    - `WAIT_TIMEOUT` – przekroczono czas oczekiwania, proces potomny nadal pracuje
    - `WAIT_FAILED` – błąd wykonania funkcji (`GetLastError()` poda kod przyczyny)
- pełny opis funkcji znajduje się tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject

{win} Funkcje `CreateProcess()` i `WaitForSingleObject()`

```
// win_createprocess.c
#include <windows.h>
#include <stdio.h>

int main(void) {
     STARTUPINFO si;
     PROCESS_INFORMATION pi[3];
     char argline[50];
     memset(&si, 0, sizeof(si));
     memset(&pi, 0, sizeof(pi));
     si.cb = sizeof(si);
     for(int i = 0; i < 3; i++) {
          sprintf(argline, "win_prog.exe Process=%d arg1 arg2", i+1);
          if(CreateProcessA(NULL,argline,NULL,NULL,0,0,"Path=C:\\My\\Path\0\0",NULL,&si,pi+i)
== 0){
               printf( "CreateProcess #%d failed (%d).\n", i+1, GetLastError() );
               return 2;
          }
          printf("Process #%d created: handle=%p, identifier=%d\n", i+1, pi[i].hProcess,
pi[i].dwProcessId);
     }
     :
```

{win} Funkcje `CreateProcess()` i `WaitForSingleObject()` c.d.

```
        :
        :
        :
    for(int i = 2; i >= 0;  i--) {
        WaitForSingleObject(pi[i].hProcess,INFINITE);
        CloseHandle( pi[i].hProcess );
        CloseHandle( pi[i].hThread );
        printf("Process #%d terminated.\n", i+1);
    }
    return 0;
}
```

{win} Natywny interfejs zarządzania procesami

Funkcja `WaitForMultipleObjects()`:

```
int WaitForMultipleObjects(
      DWORD        nCount,
       const HANDLE *lpHandles,
       BOOL         bWaitAll,
       DWORD        dwMilliseconds
);
```

- rodzic może poczekać na wielu procesów potomnych, używając funkcji `WaitForMultipleObjects()`
- argumenty określają:
  - `nCount` – liczba procesów
  - `lpHandles` – tablica z uchwytami oczekiwanych procesów
  - `bWaitAll` – jeśli różne od zera, oczekuje się zakończenia **wszystkich** procesów, jeśli równe zero – zakończenia **dowolnego**
  - `dwMilliseconds` – jak poprzednio
- wynik jak w przypadku funkcji `WaitForSingleObject()`
- pełny opis funkcji znajduje się tutaj:
  https://docs.microsoft.com/pl-pl/windows/win32/api/synchapi/nf-synchapi-waitformultipleobjects

{win} Funkcje `CreateProcess()` i `WaitForMultipleObjects()`

```
// win_createprocess_2.c
#include <windows.h>
#include <stdio.h>

int main(void) {
     STARTUPINFO si;
     PROCESS_INFORMATION pi[3];
     char argline[50];
     memset(&si, 0, sizeof(si));
     memset(&pi, 0, sizeof(pi));
     si.cb = sizeof(si);
     for(int i = 0; i < 3; i++) {
          sprintf(argline, "win_prog.exe Process=%d arg1 arg2", i+1);
          if(CreateProcessA(NULL,argline,NULL,NULL,0,0,"Path=C:\\My\\Path\0\0",NULL,&si,pi+i) == 0){
               printf( "CreateProcess #%d failed (%d).\n", i+1, GetLastError() );
               return 2;
          }
          printf("Process #%d created: handle=%p, identifier=%d\n", i+1, pi[i].hProcess,
pi[i].dwProcessId);
     }
     :
```

{win} Funkcje `CreateProcess()` i `WaitForMultipleObjects()` c.d.

```
     :
      :
      :
     HANDLE children[3] = {pi[0].hProcess,pi[1].hProcess,pi[2].hProcess};
     WaitForMultipleObjects(3, children, 1, INFINITE);
     for(int i = 2; i >= 0;  i--) {
          CloseHandle( pi[i].hProcess );
          CloseHandle( pi[i].hThread );
          printf("Process #%d terminated.\n", i+1);
     }
     return 0;
}
```

{win} Natywny interfejs zarządzania procesami

Funkcja `GetCurrentProcessId()`:

```
DWORD GetCurrentProcessId();
```

- pobranie identyfikatora (nie uchwytu!) bieżącego procesu
- funkcja zawsze kończy się pomyślnie
- pełny opis funkcji znajduje się tutaj:
  https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocessid

{win} Natywny interfejs zarządzania procesami

Funkcja `GetExitCodeProcess()`:

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);
```

- pobranie kodu zakończenia procesu o wskazanym uchwycie
- argumenty określają:
  `hProcess` - uchwyt (nie PID!) zakończonego procesu
  `lpExitCode` - wskaźnik na daną, w której funkcja umieści kod zakończenia wskazanego procesu
- funkcja zwraca wartość:
  `≠0` w przypadku poprawnego zakończenia
  `=0` w przypadku błędu (`GetLastError()` dostarczy informacji o przyczynie)
- pełny opis funkcji znajduje się tutaj:
  https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess

Fundusze Europejskie Wiedza Edukacja Rozwój | Rzeczpospolita Polska | Unia Europejska Europejski Fundusz Społeczny