Wybrane funkcje systemowe Unixa

Witold Paluszyński
witold.paluszynski@pwr.edu.pl
http://kcir.pwr.edu.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat wybranych funkcji systemowych Unixa. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Model obsługi błędów w języku C/systemie Unix

W czasie wykonywania programu mogą powstawać błędy, czyli sytuacje w oczywisty sposób nieprawidłowe. Zarówno możliwość wykrycia, i sposób reakcji na te błędy ze strony systemu operacyjnego, jak i samego programu, zależą od tego czy: błąd powstał w kodzie programu, czy też w trakcie wykonywania funkcji systemowej, oraz od tego czy jest to tzw. błąd fatalny, czy nie.

Błędy fatalne to są takie błędy, po których stan programu nie jest w pełni określony, zatem kontynuowanie pracy programu nie gwarantuje jego deterministycznego i bezpiecznego zachowania. Błędy niefatalne, dla odmiany, to takie, po których stan programu jest dokładnie znany, zatem program ogólnie może kontynuować pracę, aczkolwiek zwykle konieczne jest zaprogramowania odpowiedniej reakcji na błąd.

Przykładami błędów fatalnych są: próba wykonania nielegalnej intrukcji (kodu operacji, którego dany procesor nie obsługuje), próba odwołania się do adresu pamięci, która jest niedostępna dla procesu (albo adres spoza przestrzeni adresowej procesu, albo niedozwolony typ operacji, np. zapis do obszaru *read-only*), wykonanie niedozwolonej operacji arytmetycznej (nadmiar, lub dzielenie przez zero), itp.

Przykładami błędów niefatalnych są: niemożność otwarcia pliku, niemożność odczytu lub zapisu danych z/na pliku, niemożność przydzielenia pamięci na stercie, itp.

Obsługa błędów fatalnych w programie

Błędy fatalne są w miarę możliwości wykrywane przez system operacyjny. System Unix po wykryciu błędu fatalnego wysyła do procesu odpowiedni sygnał, dedykowany danemu rodzajowi błędu, np. `SIGILL`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, itp.

Otrzymanie sygnału normalnie powoduje natychmiastowe zatrzymanie („śmierć") procesu, ale może być przez sam proces przechwycone i obsłużone. Zwykle dobrym zwyczajem jest zrealizowanie domyślnej reakcji, aczkolwiek jest dopuszczalne przechwycenie sygnału w celu kontrolowanego zatrzymania aplikacji, poprawnego zamknięcia plików, wyświetlenia odpowiedniego komunikatu, itp.

Ma to sens w sytuacji, kiedy program jest już bardzo dobrze przygotowany i przetestowany, wszelkie wykryte błędy poprawione, i program jest przygotowywany do dystrybucji dla szerszego kręgu użytkowników nieprzygotowanych do radzenia sobie z błędami.

Błędy fatalne mogą powstawać zarówno w trakcie wykonywania kodu programu, jak i funkcji systemowych. Należy pamiętać, że nie zawsze błąd fatalny może być wykryty przez system, np. odwołanie się do niewłaściwego adresu (np. poza zakresem tablicy) często nie zostanie wykryte, zwłaszcza przy wysokim poziomie optymalizacji programu.

Obsługa błędów niefatalnych w programie

Błędy niefatalne najczęściej mogą być wykryte w trakcie wykonywania funkcji systemowych. Typową sytuacją jest, że funkcja systemowa nie może wykonać swojej operacji, efekt jej wykonania będzie inny od zamierzonego, i funkcja sygnalizuje to jako błąd. W systemie Unix istnieje tradycyjny, nieco niefortunnie stworzony mechanizm zgłaszania błędów przez funkcje systemowe.

Model sygnalizacji błędów niefatalnych przez funkcje systemowe Uniksa, i ich obsługi w programach, polega na:

1. sygnalizacji wystąpienia błędu w funkcji systemowej przez zwrócenie pewnej specyficznej wartości, która może być różna dla różnych funkcji (typowo: dla funkcji zwracających `int` jest to wartość -1, a dla funkcji zwracających wskaźnik zwykle jest to wartość `NULL`),
2. ustawienia wartości zmiennej globalnej procesu `errno` na kod zaistniałego błędu; należy pamiętać, że wartość zmiennej `errno` pozostaje ustawiona nawet gdy kolejne wywołania funkcji systemowych są poprawne.

Programista powinien zatem sprawdzić, po wywołaniu każdej funkcji systemowej, jej wartość, aby upewnić się czy zadziałała poprawnie, i podjąć właściwe działanie, gdy nie. W najprostszym przypadku można wyświetlić komunikat odpowiedni dla wartości `errno` (przydają się do tego funkcje `perror` i `strerror`) i zatrzymać program.

Ten model obsługi błędów jest niefortunny, ponieważ korzysta ze zmiennej globalnej, co jest ogólnie złą praktyką. W starszych programach jednowątkowych działało to poprawnie, ale w nowych programach tworzących wątki, korzystające ze wspólnej globalnej przestrzeni adresowej, prowadzi do niejednoznaczności. Np, gdy w jednym wątku błąd funkcji systemowej ustawi wartość `errno`, i bardzo szybko po tym w innym wątku inna funkcja również wygeneruje błąd, to nadpisze globalną wartość `errno`, być może zanim pierwszy wątek zdąży sprawdzić i odpowiednio zareagować na swój błąd.

Dygresja: istnieją jeszcze inne błędne koncepcje zagrzebane głęboko w bibliotekach funkcji uniksowych. Szereg funkcji w tych bibliotekach wykorzystuje efekty globalne, co jest tak samo fatalne jak użycie zmiennej globalnej `errno`. W programach wielowątkowych wymaga to specjalnych zabiegów, zwykle polegających na nieużywaniu tych niefortunnie wymyślonych funkcji. Ponieważ jednak w niektórych wersjach Uniksa problemy z niektórymi z tych funkcji zostały rozwiązane w implementacji, dokumentacja funkcji systemowych została uzupełniona o atrybut `MT-Safe`. Funkcje posiadające ten atrybut mogą być bezpiecznie używane w aplikacjach wielowątkowych.

Innym mechanizmem systemów uniksowych rodzącym problemy w programach wielowątkowych jest mechanizm sygnałów. Sygnały są z definicji doręczane do procesu, i jeśli ten proces ma wiele wątków to nie wiadomo jakie mają być ich reakcje.

Wszystkie te problemy zostały rozwiązane przy pomocy dodatkowych mechanizmów, jednak ich użycie wymaga zaawansowanej wiedzy programisty.

Biblioteki funkcji Uniksa

Systemy operacyjne typowo posiadają zbiory funkcji przydatnych w różnego rodzaju obliczeniach, zwane **bibliotekami funkcji**, a funkcje w nich zawarte zwane są **funkcjami bibliotecznymi**.

W systemie Unix takie biblioteki istnieją w postaci binarnej i we współczesnych systemach najczęściej występują w dwóch wersjach: biblioteki **linkowanej statycznie** albo **linkowanej dynamicznie**, często określanej również jako **współdzielonej**.

Wykorzystanie bibliotek dynamicznych ma szereg zalet i we współczesnych systemach jest najczęściej opcją domyślną. Linkowanie w czasie kompilacji nie powoduje dopisania do programu kodu binarnego funkcji bibliotecznych, tylko informacji o tych funkcjach i parametrach biblioteki. Taki program zapisany w wersji binarnej w pliku na dysku typowo zajmuje mniej miejsca, ale w czasie uruchamiania musi skorzystać z linkera dynamicznego, który załaduje do pamięci RAM razem z kodem programu kod wszystkich wymaganych przezeń funkcji bibliotecznych. Z kolei, gdy w systemie uruchamianych jest więcej programów korzystających z danej biblioteki dynamicznej, może ona być załadowana tylko raz, i współdzielona przez wszystkie te programy.

Jednak uruchomienie programu zlinkowanego dynamicznie na innym systemie jest możliwe tylko wtedy, gdy w tym drugim systemie istnieje wersja tej samej biblioteki.

Biblioteki linkowane statycznie są historycznie wcześniejsze. Wykorzystanie w czasie kompilacji programu biblioteki linkowanej statycznie powoduje dopisanie do programu binarnego kodu wszystkich używanych w nim funkcji bibliotecznych, i taki program binarny jest większy, czasami znacznie, ale może być wykonywany również w innym systemie, który danej biblioteki nie posiada. Z kolei, jeśli na docelowym systemie byłoby wykonywanych wiele programów statycznie zlinkowanych z tą biblioteką, to musiałyby one wszystkie być niezależnie załadowane do pamięci RAM, zajmując znacznie więcej miejsca, niż przy linkowaniu bibliotek współdzielonych.

Na przykład, istnieje tzw. biblioteka matematyczna, udostępniająca takie funkcje jak `sin` lub `sqrt`, obliczające ich wartości ogólnie znanymi algorytmami. Ich dostępność w systemie powoduje, że programista nie musi ich każdorazowo sam implementować, ale mógłby to zrobić gdyby taka biblioteka nie istniała.

Biblioteki funkcji odgrywają dużą rolę we współczesnych systemach. Istnieje ich bardzo dużo i wiele programów z nich korzysta. Przykładem mogą być programy okienkowe, które we współczesnych systemach często wykorzystują setki bibliotek dynamicznych.

Funkcje regcomp(3C)

Funkcje regcomp(3C) realizują dopasowanie napisów do wzorców (opisanych na stronie manuala regex(5)) typu BRE (*Basic Regular Expressions*) oraz ERE (*Extended Regular Expressions*), tych drugich głównie różniących się od pierwszych obsługą alternatywy |, i brakiem obsługi odwołań wstecznych typu \1.

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);
int regexec(const regex_t *preg, const char *string, size_t nmatch,
            regmatch_t pmatch[], int eflags);
size_t regerror(int errcode, const regex_t *preg, char *errbuf,
            size_t errbuf_size);
void regfree(regex_t *preg);
```

Struktura regmatch_t zawiera między innymi pola: rm_so i rm_eo typu regoff_t wypełniane przez funkcję regexec wartościami offsetu od początku bufora dopasowywanego stringu do początku znalezionego dopasowania, oraz do pierwszego znaku po ostatnim znaku dopasowania.

Funkcja sprawdzająca dopasowanie napisu do wzorca (przykład z manuala):

```
#include <regex.h>

int match(const char *string, char *pattern)
/*
 * Match string against the extended regular expression in pattern,
 * treating errors as no match.
 *
 * return 1 for match, 0 for no match
 */
{
  int status;
  regex_t re;
  if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
    return(0);                          /* report error */
  }
  status = regexec(&re, string, (size_t) 0, NULL, 0);
  regfree(&re);

  if (status != 0) {
    return(0);                          /* report error */
  }
  return(1);
}
```

Funkcje systemowe Uniksa

Niezależnie od funkcji bibliotecznych, praktycznie w każdym systemie operacyjnym istnieje pewna grupa funkcji realizowanych przez jądro systemu i udostępnianych aplikacjom (procesom). Zwane **funkcjami systemowymi** lub **wywołaniami systemowymi** (*system calls*) realizują operacje, których procesy nie mogą zrealizować sobie same, natomiast system operacyjny — jako uprzywilejowany gospodarz systemu — może zrealizować i udostępnia procesom w ramach świadczonych im usług.

Można ogólnie wymienić następujące grupy (i przykłady) funkcji systemowych Uniksa:

- funkcje wejścia/wyjścia (`read`),
- funkcje obsługi procesów (`fork`),
- funkcje komunikacji międzyprocesowej i synchronizacji (`msgget`),
- funkcje zarządzania i sterowania urządzeniami (`fcntl`),
- funkcje tworzenia i zarządzania zabezpieczeniami (`setuid`),
- funkcje obsługi systemu plików (`unlink`),
- funkcje informacyjne (`time`).

Te grupy nie są rozłączne; pewne funkcje systemowe można zaliczyć do dwóch lub więcej z nich. Na przykład, `write` jest funkcją wejścia/wyjścia i komunikacji międzyprocesowej, a `chmod` (ustawiająca prawa dostępu do pliku dyskowego) jest zarazem funkcją obsługi systemu plików, jak i zarządzania zabezpieczeniami.

Informacje o pliku: `inode`

System Unix przechowuje informacje o wszystkich plikach systemu plików w blokach kontrolnych *i-node* (węzeł informacyjny, czyli i-węzeł). Dostęp do tych bloków jest możliwy za pośrednictwem struktury `stat`:

```
struct  stat {
  dev_t    st_dev;      /* ID of device containing */
                        /* a directory entry for this file */
  dev_t    st_rdev;     /* ID of device */
                        /* This entry is defined only for */
                        /* char special or block special files */
  ino_t    st_ino;      /* Inode number */
  mode_t   st_mode;     /* File mode (see mknod(2)) */
  nlink_t  st_nlink;    /* Number of links to file */
  uid_t    st_uid;      /* User ID of the file's owner */
  gid_t    st_gid;      /* Group ID of the file's group */
  off_t    st_size;     /* File size in bytes */
  time_t   st_atime;    /* Time of last access */
  time_t   st_mtime;    /* Time of last data modification */
  time_t   st_ctime;    /* Time of last file status change */
                        /* Times measured in seconds since */
                        /* 00:00:00 UTC, Jan. 1, 1970 */
  long     st_blksize;  /* Preferred I/O block size */
  blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
  ...
};
```

Odczyt struktury `stat` jest możliwy za pomocą poniższych funkcji:

```
int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

To znaczy, strukturę `stat` można odczytać dla pliku podając jego nazwę w katalogu na dysku (funkcja `stat`), albo podając numer deskryptora otwartego pliku, niezależnie od sposobu w jaki został otwarty (funkcja `fstat`).

Dla linków symbolicznych funkcja `stat` odczytuje strukturę `stat` docelowego pliku. Istnieje też możliwość odczytania struktury `stat` samego pliku linku (funkcja `lstat`).

Zwróćmy uwagę, że w strukturze `stat` nie ma nazwy pliku. Nazwa pliku nie jest cechą pliku, tylko jest związana z pozycją w katalogu systemu plików, za pośrednictwem której można uzyskać dostęp do pliku. Ta pozycja nazywana jest linkiem twardym pliku. Jeden plik może mieć więcej niż jeden link twardy, i różne linki twarde do danego pliku mogą znajdować się w różnych katalogach, i mieć różne nazwy.

W odróżnieniu, linki symboliczne nie są dodatkowymi linkami do danego pliku, tylko są oddzielnymi specjalnymi plikami, które odwołują się do danego pliku przez nazwę, a dokładniej przez ścieżkę (względną lub bezwzględną) w systemie plików. Zatem może istnieć link symboliczny do pliku, który nie istnieje, ale nie może istnieć link twardy do pliku, który nie istnieje (może się to zdarzyć gdy system plików ulegnie uszkodzeniu).

Inne operacje na plikach i deskryptorach plików

Istnieje szereg funkcji systemowych wykonujących operacje na plikach w systemie plików, oraz na deskryptorach plików otwartych przez program:

`rename` — zmienia nazwę pliku

`chmod` — zmienia prawa dostępu do pliku

`umask` — ustawia maskę praw dostępu nowo tworzonych plików

`access` — sprawdza prawa dostępu procesu do pliku

`link` — tworzy nowy link (twardy) do pliku

`symlink` — tworzy nowy link symboliczny do pliku

`unlink` — kasuje plik (lub link symboliczny)

`remove` — kasuje plik lub pusty katalog

`rmdir` — kasuje katalog (musi być pusty)

`mkdir` — tworzy katalog

`chdir` — przełącza się do innego katalogu

`dup` — duplikuje deskryptor otwartego pliku

`fcntl` — pozwala wykonać wiele różnych operacji na deskryptorze otwartego pliku

Czytanie zawartości katalogów dyskowych

Poruszanie się po dyskowym systemie plików niezależne od jego typu jest możliwe za pomocą funkcji `opendir` i `readdir`, które zwracają wskaźnik do struktury, odpowiednio `DIR` i `dirent` (tylko treść tej ostatniej jest ciekawa i jest tu pokazana):

```
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

```
struct  dirent {
        ino_t           d_ino;
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
```

Odczyt zawartości katalogu dyskowych wymaga otwarcia pliku katalogu funkcją `opendir`. Tworzy ona strukturę o nazwie `DIR`, i zwraca do niej wskaźnik, który można odczytywać wielokrotnie funkcją `readdir`. Każde jej wywołanie zwraca wskaźnik do struktury `dirent` opisującej kolejną pozycję katalogu (plik zwykły lub specjalny, albo podkatalog). Po wyczerpaniu wszystkich pozycji katalogu `readdir` zwraca `NULL`.

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

DIR *dirp;
struct dirent *direntp;
```

```
dirp = opendir(".");
while ((direntp = readdir( dirp )) != NULL)
  (void)printf("%s\n", direntp->d_name);
(void)closedir(dirp);
```

Przykład: wyświetlanie struktury katalogów

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void printdir(char *dir, int depth) {
  DIR *dp;
  struct dirent *entry;
  struct stat statbuf;

  if((dp = opendir(dir)) == NULL) {
    fprintf(stderr,"cannot open directory: %s\n", dir);
    return;
  }
  chdir(dir);
  while((entry = readdir(dp)) != NULL) {
    stat(entry->d_name,&statbuf);
    if(S_ISDIR(statbuf.st_mode)) {
      /* Found a directory, but ignore . and .. */
      if(strcmp(".",entry->d_name) == 0 ||
         strcmp("..",entry->d_name) == 0)
           continue;
```

```
      printf("%*s%s/\n",depth,"",entry->d_name);
      /* Recurse at a new indent level */
      printdir(entry->d_name,depth+4);
    }
    else printf("%*s%s\n",depth,"",entry->d_name);
  }
  chdir("..");
  closedir(dp);
}
```

Funkcje czasu zegarowego

Funkcja `time` zwraca czas bieżący w postaci liczby sekund, jakie upłynęły od godziny 0:00 dnia 1 stycznia 1970:

```
#include <sys/types.h>
#include <time.h>

time_t time(time_t *tloc);
```

Unixowe funkcje czasu zegarowego obsługują czas od godziny 0:00 1 stycznia 1970 do godziny 04:14:07 19 stycznia 2038. Wynika to z faktu, że typ `time_t` jest typem liczby całkowitej `long` (32-bitowej) ze znakiem, więc o zakresie wartości dodatnich około 2 miliardów.

Ten system reprezentacji czasu zwany jest **czasem uniksowym** lub **czasem Posiksowym**, i jest stosowany w szeregu innych systemów operacyjnych (ale nie w Windows). Jedną z jego cech jest sekundowa dokładność: zarówno odczyt czasu rzeczywistego (funkcja `time`), zawieszanie procesu na określony czas (funkcja `sleep`), programowanie timera procesowego (funkcja `alarm`), jak i szereg funkcji wyświetlania wartości czasowych, obejmują całkowitosekundowe wartości czasu. Operowanie wartościami czasowymi o większej dokładności wprowadzono nowszymi standardami POSIX i stanowią one uzupełnienie współistniejące z tradycyjnymi funkcjami czasu.

Funkcje czasu — napisy sformatowane

Funkcja `ctime` tworzy zapis daty i czasu w postaci stringa o ustalonym 26-znakowym formacie: `"Thu Nov 23 11:04:20 2000\n\0"`. Wyświetlany jest zawsze czas lokalny, i napis ten nie podlega żadnym, lokalizacjom, konwencjom, ani konwersjom.

```
#include <time.h>

char *ctime(const time_t *clock);
```

Istnieje również rodzina funkcji do tworzenia dowolnie sformatowanych napisów czasowych, z uwzględnieniem lokalizacji (języka i konwencji lokalnych):

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize, const char
                *restrict format, const struct tm *restrict timeptr);

int cftime(char *s, char *format, const time_t *clock);

int ascftime(char *s, const char *format, const struct tm *timeptr);
```

Funkcje czasu — obliczenia kalendarzowe

```
struct tm *localtime(const time_t *clock);
time_t mktime(struct tm *timeptr);

struct  tm {
   int  tm_sec;    /* seconds after the minute - [0, 61] */
                   /* for leap seconds */
   int  tm_min;    /* minutes after the hour - [0, 59] */
   int  tm_hour;   /* hour since midnight - [0, 23] */
   int  tm_mday;   /* day of the month - [1, 31] */
   int  tm_mon;    /* months since January - [0, 11] */
   int  tm_year;   /* years since 1900 */
   int  tm_wday;   /* days since Sunday - [0, 6] */
   int  tm_yday;   /* days since January 1 - [0, 365] */
   int  tm_isdst;  /* flag for alternate daylight savings time */
};
```

Funkcja `localtime` tworzy i wypełnia strukturę `tm`, która daje dostęp do elementów aktualnego czasu. Brana jest pod uwagę lokalna strefa czasowa, czas letni/zimowy, lata przestępne, a nawet sekundy przestępne.[1]

Funkcja `mktime` zamienia strukturę czasową `tm` na liczbę sekund jak w funkcji `time`, dodatkowo kompletując i normalizując pola w strukturze, które mogą być wypełnione częściowo, lub poza zakresem (np. `tm_hour` < 0 lub > 23).

[1] Więcej o sekundach przestępnych: `http://queue.acm.org/detail.cfm?id=1967009`

Timer procesu

Interfejs tradycyjny wprowadził własny timer programowy czasu rzeczywistego dla każdego procesu. Nazywany **budzikiem** (ang. *alarm*) timer programowany jest w sekundach, i po przeterminowaniu przysyła do procesu dedykowany mu sygnał SIGALRM. [2]

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Nie ma oddzielnych operacji zaprogramowania i wystartowania timera — po zaprogramowaniu niezerowej wartości timer od razu uruchamia się. Jeśli był już uruchomiony to wywołanie funkcji powoduje jego zaprogramowanie na nową wartość. W tym przypadku funkcja zwraca liczbę sekund pozostałą do poprzednio zaprogramowanego przeterminowania. Wywołanie funkcji z argumentem 0 powoduje zatrzymanie timera, o ile był uruchomiony.

[2] Fakt, że twórcy Uniksa uznali, że odmierzanie czasu rzeczywistego dla procesu może być wyrażone w sekundach, jest swoistym znakiem czasu. Na początku lat 70-tych dwudziestego wieku nie przewidywali oni zastosowań, w których potrzebne (albo wręcz praktycznie możliwe) byłoby odcinki 0.1 sekundy, 0.01 sekundy, albo nawet na milisekundy, mikrosekundy, nanosekundy ...

Zawieszenie wykonywania procesu — funkcja `sleep`

W tradycyjnym interfejsie systemów uniksowych istnieje funkcja `sleep` pozwalająca zawiesić wykonywanie procesu na określoną liczbę sekund. W trakcie wykonywania tej funkcji proces pozostaje w stanie uśpienia, normalnie wykorzystywanym do oczekiwania na jakieś zasoby, blokady, operacje I/O, itp.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Funkcja `sleep` może zakończyć się po czasie innym niż zadana liczba sekund. Szybszy powrót jest możliwy gdy proces otrzyma sygnał, nawet jeśli zostanie on obsłużony — po zakończeniu obsługi sygnału i wznowieniu funkcji `sleep` następuje natychmiastowe jej zakończenie. W takim przypadku funkcja zwraca liczbę „nieprzespanych" sekund. Możliwy jest również późniejszy niż zadany powrót z funkcji `sleep`, np. w wyniku zwykłego planowania procesów.

Niektóre starsze implementacje funkcji `sleep` wykorzystywały sygnał SIGALRM, i w efekcie kolidowały z ewentualnym wykorzystaniem timera przez proces. Współczesne wersje nie mają tej wady. Działają również poprawnie w środowisku wielowątkowym, usypiając tylko wywołujący wątek.

Funkcje czasu wirtualnego procesów

Czas obliczeń CPU procesów czasem nazywa się w terminologii systemów uniksowych **czasem wirtualnym**. Obliczanie tego czasu obejmują zupełnie inne zasady niż dla czasu rzeczywistego (kalendarzowego): różnią się jednostki, wartości odniesienia, oraz struktury danych w jakiej jest on przekazywany.

```
#include <sys/times.h>
#include <limits.h>

clock_t times(struct tms *buf);

struct tms {
   clock_t tms_utime;     /* user time */
   clock_t tms_stime;     /* system time */
   clock_t tms_cutime;    /* user time, children */
   clock_t tms_cstime;    /* system time, children */
};
```

- Funkcja `times` zwraca czas obliczeń od arbitralnie ustalonego momentu w czasie (może to być np. moment startu systemu); jednostką jest tzw. *tick*, którego liczbę na sekundę określa makro `clk_tck` (przykładowo 50, 60, albo 100).

- Czas procesora zużyty przez proces liczony jest w rozbiciu na tzw. czas użytkownika, czyli instrukcje programu, i czas systemowy, tzn. czas zużyty na obliczenia w ramach wywołanych przez program funkcji systemowych.

- Podana struktura jest wypełniana przez funkcję `times` wartościami czasu procesora zużytego przez proces i jego podprocesy, które już się zakończyły i zostały poprawnie obsłużone funkcją `wait`. Wartości czasu wirtualnego podprocesów są podobnie liczone od arbitralnego momentu i podane w tych samych jednostkach w rozbiciu na czas użytkownika i czas systemowy.

Timery programowe POSIX TMR

Inna specyfikacja standardu POSIX, zwana TMR, wprowadza inny rodzaj timerów. Są one tworzone w programie w powiązaniu z istniejącymi zegarami czasu rzeczywistego (jak `CLOCK_REALTIME`). Program może stworzyć wiele takich timerów, i należą one do danego procesu (nie są dziedziczone przez podprocesy).

Tworzenie timerów POSIX TMR przebiega według następującego schematu:

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

int timer_create(clockid_t clock_id, struct sigevent *restrict evp,
                 timer_t *restrict timerid);
```

Struktura `struct sigevent` określa dla danego timera jaki sygnał ma być wysłany w momencie przeterminowania (domyślnym jest sygnał `SIGALRM` co pozwala na całkowite pominięcie struktury `struct sigevent` w wywołaniu `timer_create`). Inną możliwością powiadamiania procesu o przeterminowaniu timera jest uruchomienie wątku. Możliwe jest również żądanie braku jakiegokolwiek powiadomienia. W takim przypadku timer w czasie pracy musi być każdorazowo odpytywany o pozostały czas.

Opcje powiadamiania dla timerów POSIX TMR

```
struct sigevent {
  int    sigev_notify;          /* notification type */
  int    sigev_signo;           /* signal number */
  union  sigval sigev_value;    /* signal value */
  ...
};
union sigval {
  int    sival_int;             /* integer value */
  void   *sival_ptr;            /* pointer value */
};
```

Rodzaj powiadamiania związany z timerem określa się w strukturze `sigevent` tworząc dany timer. Wartość `sigev_notify` może przybierać następujące wartości:

SIGEV_NONE — brak powiadomienia

SIGEV_SIGNAL — zwykłe powiadamianie sygnałem

SIGEV_THREAD — w momencie przeterminowania timera uruchom wątek

Funkcje timerów POSIX TMR

Timery POSIX TMR programuje się i uruchamia funkcją `timer_settime`, a pozostały czas odczytuje funkcją `timer_gettime`, analogicznie jak dla timerów XSI. Funkcja `timer_settime` posiada opcjonalne flagi, i m.in. może pracować z czasem bezwzględnym (flaga `TIMER_ABSTIME`, wymaga zaprogramowania pełnego czasu zegarowego, zamiast interwału czasowego). Pozwala to kontrolować i korygować dryf timera w programach (patrz poniżej).

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value, struct itimerspec *ovalue);
int timer_delete(timer_t timerid);
```

Funkcja `timer_getoverrun` zwraca liczbę przeterminowań danego timera, dla których nie został doręczony sygnał wskutek wstrzymywania. Proces może czasowo wstrzymać otrzymywanie sygnałów, wtedy kolejne przeterminowania timera nie generują dalszych sygnałów, natomiast proces może dowiedzieć się o takiej sytuacji dzięki tej funkcji.

Wartości czasowe dla timerów POSIX TMR

Czasy do programowania timerów POSIX TMR określa się za pomocą struktur `struct itimerspec`, które analogicznie do struktur `struct itimerval` timerów XSI, zawierają co najmniej pola:

```
struct timespec it_interval;  /* timer period */
struct timespec it_value;     /* timer expiration */
```

Wartości czasowe wykorzystywane przez timery POSIX TMR są nieco inne niż dla timerów POSIX XSI. Struktura `struct timespec` zawiera co najmniej następujące elementy, pozwalające wyznaczać czas jako kombinację liczby sekund i nanosekund:

```
time_t  tv_sec;   /* seconds */
long    tv_nsec;  /* nanoseconds */
```

Badanie timera POSIX TMR programem periodicmessage.c
Badanie czasu wykonywania funkcji programem tmrtimer.c

Zegary programowe czasu rzeczywistego

Standard POSIX specyfikacja TMR wprowadziła mechanizm zegara pozwalającego obliczać czas kalendarzowy z dokładnością większą od jednej sekundy. Następujące funkcje wykonują operacje na zegarach:

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

Parametr `clock_id` określa zegar, na którym ma być wykonana dana operacja, przy czym każdy system obowiązkowo musi implementować zegar czasu rzeczywistego `CLOCK_REALTIME`.

Jak widać, wartości czasowe wykorzystywane przez te zegary są takie same jak dla timerów POSIX TMR.

Zawieszenie wykonywania wątku — funkcja `nanosleep`

Rozszerzenie *realtime* standardu POSIX wprowadziło kolejną funkcję zawieszania procesu (dokładniej: wątku) na określony czas, wyrażony za pomocą tej samej struktury `timespec`:

```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

Podobnie jak inne funkcje zawieszające procesy, funkcja wraca po upłynięciu zadanego czasu, lub nieco później, w przypadku:

- zaokrąglenia wartości czasowej do rozdzielczości
- planowania procesów i obciążenia systemu

W przypadku otrzymania (i obsłużenia) sygnału funkcja natychmiast kończy pracę, sygnalizując powrót przed upływem zadanego czasu. W takim przypadku, o ile drugi argument nie jest NULL, funkcja wpisuje do tej struktury wartość pozostałego (nieprzespanego) czasu.

Badanie rozdzielczości spania funkcji `nanosleep`.

Zawieszenie wykonywania wątku — inne możliwości

Często pojawia się potrzeba czasowego wstrzymania wykonywania procesu lub wątku. Poza wykonaniem tej operacji za pomocą funkcji `sleep()` i `nanosleep()` przedstawionych wyżej istnieje szereg innych możliwości, które są czasami wygodne.

Możliwe jest wykorzystanie timera do obudzenia sygnałem i zawieszenie procesu na czas nieokreślony (funkcje `pause()`, `sigpause()`, i inne). Do programowania doręczenia sygnału jest dedykowany sygnał `SIGALRM` i funkcja `alarm()`, ale timery TMR dostarczają wielu alternatywnych możliwości.

Inną możliwością jest *polling*, czyli odpytywanie jakiegoś timera lub zegara. Proces lub wątek może w pętli zawieszać się na jakieś krótkie odcinki czasu, okresowo sprawdzając upływ czasu, przed wznowieniem normalnej pracy. Powoduje to co prawda pewne obciążenie procesora w tym okresie „zawieszenia", ale pozwala wątkowi kontrolować sytuację i reagować na inne zdarzenia.

Również możliwe jest wykorzystanie innych mechanizmów, niezwiązanych z odmierzaniem czasu, na przykład blokady pliku, semafora, muteksa, bariery, itp. Konkretne rozwiązanie może być właściwe, jeżeli zawieszenie wątku jest związane z odpowiednimi operacjami, a niekoniecznie z określoną długością odcinka czasu.