Czas systemowy

System operacyjny oferuje szereg usług związanych z czasem. Jedną z podstawowych jest informacja o aktualnym czasie bezwzględnym, zwanym **czasem rzeczywistym**. Czas rzeczywisty nazywany jest również czasem kalendarzowym i wyrażany jest względem pewnej strefy czasowej.

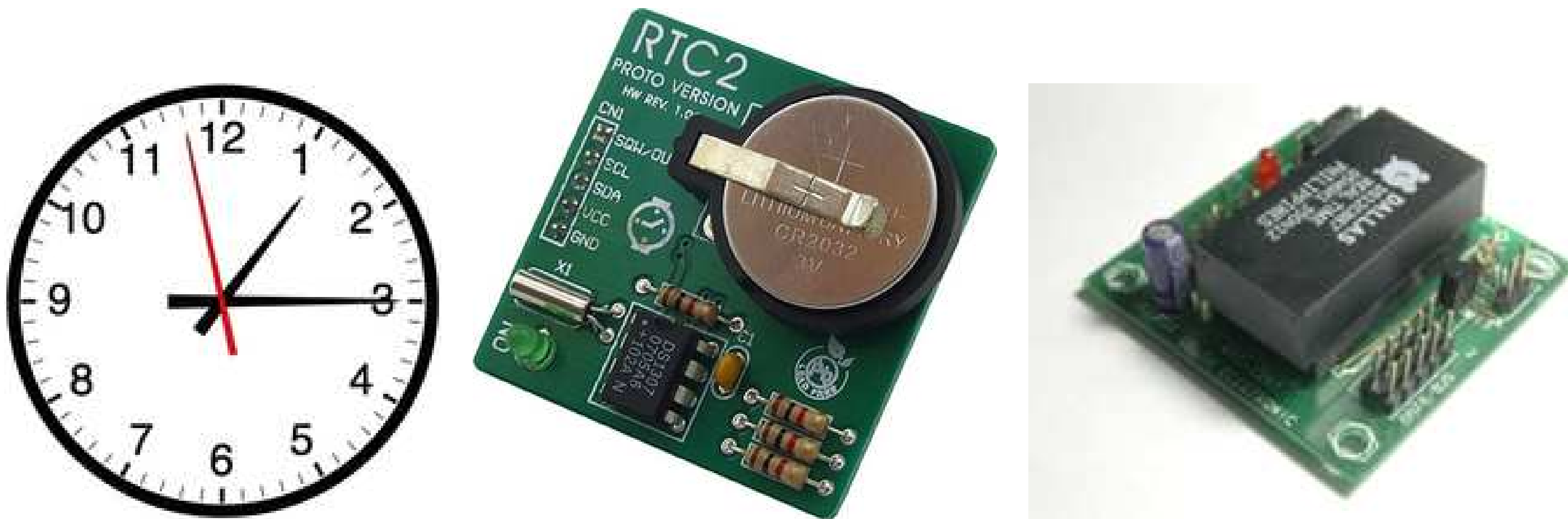
System odmierza upływający czas na bieżąco, za pomocą urządzenia programowego zwanego **zegarem**, zliczając kolejne odcinki czasu na podstawie przerwań ze sprzętowego **timera**. Jednak problemem jest początkowa **inicjalizacja** zegara.

Dodatkowo, problemem może być dokładność mierzonego czasu. Po pewnym czasie pracy wskazania każdego zegara stają się niedokładne. Potrzebna jest okresowa **synchronizacja** zegara z jakimś wzorcem czasu.

Poza informacją o aktualnym czasie rzeczywistym system operacyjny świadczy jeszcze inne usługi związane z czasem, takie jak: zawieszanie procesu/wątku na określony czas, usługi timerów, i inne.

Sprzętowy zegar czasu rzeczywistego RTC

Zegar czasu rzeczywistego (*real time clock RTC*) jest urządzeniem odmierzającym upływający czas w skali bezwzględnej. Zegar RTC typowo posiada własne źródło zasilania, albo zewnętrzne albo wbudowane w układ zegara, pozwalające na ciągłe odmierzanie czasu przy braku stałego zasilania.



Sprzętowy zegar czasu rzeczywistego z własnym zasilaniem jest typowym wzorcem czasu stosowanym w systemach komputerowych w celu umożliwienia inicjalizacji zegara systemowego po starcie systemu. Jednak dokładność zegarów RTC jest często niewystarczająca dla normalnej pracy systemu.

Synchronizacja zegara systemowego

Dla zapewnienia dokładnego pomiaru czasu, system po starcie może zsynchronizować swój zegar z jakimś dokładnym wzorcem zewnętrznym. Na przykład, może to być internetowy serwer czasu. Jednak wiele systemów czasu rzeczywistego z różnych powodów nie może być połączonych z Internetem. W takich systemach czasami konieczne jest korzystanie z dokładnego wzorca czasu, takiego jak np. zegar atomowy.

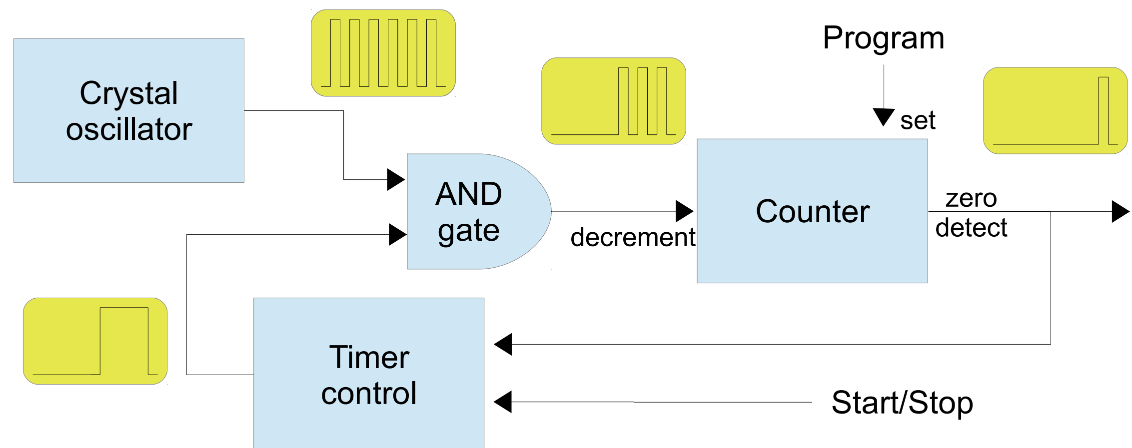
W systemach czasu rzeczywistego często ważniejsza od dokładności bezwzględnej jest synchronizacja między elementami systemu. Rozważmy na przykład linię produkcyjną o prędkości wielu metrów na sekundę, w którym systemy obsługujące różne elementy linii miałyby czas różniący się o sekundę, lub więcej. Innym dobrym przykładem jest konieczność synchronizacji zegarów krytycznych systemów mikroprocesorowych zainstalowanych na pokładzie współczesnego samolotu pasażerskiego. W takich systemach można desygnować jeden z systemów jako wzorcowy, i pozostałe systemy powinny z nim okresowo synchronizować swój czas.

W czasie startu systemu zegar jest inicjalizowany skokowo. Jednak okresowa synchronizacja, w czasie normalnej pracy systemu, musi być wykonywana bardzo małymi kroczkami, aby nie spowodować zaburzeń w pracy systemu.

Dodatkowo, po dokładnym zsynchronizowaniu swojego zegara programowego, system operacyjny może/powinien dokonać synchronizacji zegara sprzętowego RTC.

Timer sprzętowy

Timer (czytaj: tajmer) jest urządzeniem odmierzającym odcinek czasu, po którym generuje zdarzenie. W odróżnieniu od zegarów które typowo liczą czas w sposób ciągły „do przodu”, timery zwykle odmierzają odcinek czasu od zaprogramowanej wartości w dół, i generują jakieś zdarzenie po osiągnięciu wartości zero.



Elektroniczny timer sprzętowy jest skonstruowany z oscylatora, który generuje przebieg zmienny o określonym okresie i licznika zaprogramowanego na pewną liczbę, który przy każdym impulsie oscylatora dekrementuje (zmniejsza o jeden) tę wartość. Po osiągnięciu zera timer generuje sygnał i cykl kończy się.

Timer sprzętowy (cd.)

Osiągnięcie zera przez licznik nazywamy **przeterminowaniem** (*expiration*) timera. Sytuację tę timer sygnalizuje generując jakieś zdarzenie. W systemie komputerowym często jest to **przerwanie**.

Timer elektroniczny może pracować cyklicznie, automatycznie wznawiając odliczanie po przeterminowaniu się.

System komputerowy zwykle posiada jeden lub więcej timerów sprzętowych, natomiast system operacyjny może implementować wiele timerów programowych.

System operacyjny może utrzymywać informację o czasie rzeczywistym (systemowym) przez inkrementowanie wartości czasu po każdym przeterminowaniu timera.

Funkcje czasu — interfejs tradycyjny

Tradycyjne uniksowe funkcje czasu zegarowego wyrażają czas w sekundach. Funkcja `time` zwraca czas bieżący jako liczbę sekund, jakie upłynęły od godziny 00:00 dnia 1 stycznia 1970.

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

time_t time(time_t *tloc);
```

Typ `time_t` jest równoważny `long int` (ze znakiem) i pozwala na reprezentację czasu od godziny 0:00 1 stycznia 1970 do godziny 04:14:07 19 stycznia 2038.[1]

[1] Fakt że do odliczania czasu użyta jest liczba ze znakiem wydaje się marnotrawstwem jednego cennego bitu. Użycie liczby bez znaku „wydłużyłoby życie" Uniksa do roku 2106 (ale kosztem niemożności zgłaszania błędu funkcji `time` przez wartość -1). Zatem w roku 2038 można oczekiwać w systemach komputerowych na platformie Uniksa problemów z czasem, podobnych do tych, które występowały na innych platformach na początku roku 2001. W rzeczywistości różne problemy z czasem zaczęły się już pojawiać. 10 stycznia 2004 o godzinie 14:37:04 minęła połowa okresu „życia" Uniksa (czyli ustawił się najwyższy bit), ale właśnie dzięki zastosowaniu liczby ze znakiem obyło się bez większych błędów, zawinionych przez programistów, którzy by o tym znaku zapomnieli. 12 maja 2006 pojawiły się raporty o wielu „zwisach" baz danych, które nastąpiły dokładnie jeden miliard sekund przed feralną datą 2038 roku. Okazało się, że w niektórych serwerach ustawione były tak długie time-outy na transakcje, i programy sprawdzające datę po tym okresie nie mogły sobie poradzić z otrzymanymi wynikami...

Funkcje czasu — obliczenia kalendarzowe

Funkcja `localtime` tworzy i wypełnia strukturę `struct tm`, która daje dostęp do elementów aktualnego czasu. Brana jest pod uwagę lokalna strefa czasowa, czas letni/zimowy, lata przestępne, a nawet sekundy przestępne.[2]

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).

```
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 */
};
```

[2] Ostatnia sekunda przestępna, określana przez organizację IERS (International Earth Rotation Service), wystąpiła (równocześnie na całym świecie): 2016-12-31 23:59:60Z. Więcej o sekundach przestępnych: http://queue.acm.org/detail.cfm?id=1967009

Funkcje czasu wirtualnego procesów

Zupełnie inną rolę pełni funkcja `times` obliczająca czas procesora zużyty na obliczenia danego procesu.

```
#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 zegarowy, jaki upłynął od jakiegoś arbitralnie ustalonego momentu w czasie. (Może to być np. moment startu systemu.) Jednostką jest tzw. **tik** (*tick*), którego liczbę na sekundę określa makro CLK_TCK (przykładowo 50, 60, a obecnie najczęściej 100 lub 1000).

- Struktura `struct tms` jest wypełniana przez funkcję `times` wartościami czasu procesora zużytego przez proces, i oddzielnie jego zakończone podprocesy, które zostały poprawnie obsłużone funkcją `wait`. Te wartości czasu są podobnie liczone od arbitralnego momentu i podane w tych samych jednostkach.

- Ponadto wirtualne czasy zarówno procesu jak i potomków liczone są w rozbiciu na tzw. czas użytkownika, czyli wykonanie instrukcji programu, i czas systemu, tzn. obliczenia jądra Uniksa: funkcje systemowe i operacje pomocnicze (narzuty).

Poza rolą jednostki czasu wirtualnego procesów, tik tradycyjnie pełni w systemach operacyjnych inną ważną rolę. Tikiem nazywane jest przerwanie zegarowe (ściślej, tik jest okresem tego przerwania, typowo 100x/s lub 1000x/s, czyli 10ms lub 1ms), obsługiwane przez system, który budzi się z tą częstotliwością, i obsługuje różne zdarzenia: przeterminowane timery (systemowe), planowanie procesów, itp. Wartość tik definiuje zatem **rozdzielczość** (*resolution*) zegara systemowego, czyli częstotliwość z jaką jest aktualizowany.

W nowoczesnych systemach operacyjnych ta rola tiku coraz częściej okazuje się nieodpowiednia. Dla systemów czasu rzeczywistego, planowanie procesów z okresem 10 milisekund jest często niewystarczające. Natomiast w systemach wymagających energooszczędności, budzenie się 100 razy na sekundę uniemożliwia procesorowi wchodzenie w tryby głębokiego uśpienia.

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. [3]

```
#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.

[3] 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 — 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);
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jaka jest podstawowa jednostka czasu zegara czasu rzeczywistego systemów UNIX-owych i POSIX-owych, i jaki czas absolutny mierzy ten zegar?

2. Na czym polega proces synchronizacji czasu przez system operacyjny?

3. Czym różni się czas rzeczywisty od czasu wirtualnego?

4. Czym różnią się zegary od timerów?

Timery programowe POSIX XSI

Starsza, bardziej rozpowszechniona i szerzej implementowana specyfikacja standardu POSIX, zwana XSI (*X/Open System Interface*) wymaga, żeby każda implementacja dostarczała co najmniej następujących timerów:

`ITIMER_REAL` — odmierza w czasie rzeczywistym i w momencie przeterminowania generuje sygnał `SIGALRM`.

`ITIMER_VIRTUAL` — odmierza w czasie wirtualnym (czasie CPU procesu) i w momencie przeterminowania generuje sygnał `SIGVTALRM`.

`ITIMER_PROF` — odmierza w czasie wirtualnym (czasie CPU i czasie systemowym procesu) i w momencie przeterminowania generuje sygnał `SIGPROF`.

Do zarządzania timerami służą poniższe funkcje, gdzie parametr `which` określa konkretny timer, parametr `value` określa wartość ustawianego czasu, a parametr `ovalue` funkcja `setitimer` ustawia na pozostałą część poprzedniej wartości.

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

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
                              struct itimerval *restrict ovalue);
```

Wartości czasowe dla timerów POSIX XSI

Do programowania timerów XSI wykorzystywana jest struktura `struct itimerval` zawierająca co najmniej następujące elementy:

```
struct timeval it_value;    /* time until next expiration */
struct timeval it_interval; /* value to reload into the timer */
```

Oba powyższe elementy określane są za pomocą struktury czasowej `struct timeval` zawierającej co najmniej następujące elementy:

```
time_t tv_sec;              /* seconds since the Epoch */
time_t tv_usec;             /* and microseconds */
```

Jeśli pole `it_interval` wartości `*value` jest różne od zera to timer jest restartowany natychmiast po przeterminowaniu. Jeśli pole `it_value` wartości `*value` wynosi 0 to funkcja `setitimer` zatrzymuje timer.

Badanie timera XSI programem periodicasterisk.c
Badanie czasu wykonywania funkcji programem xsitimer.c
Badanie rozdzielczości funkcji `nanosleep` programem nanotest.c

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.

Ćwiczenia z zegarami czasu rzeczywistego[4]

Pomiar czasu wywołania funkcji za pomocą zegara `CLOCK_REALTIME` za pomocą programu clockrealtimetiming.c

Pomiar czasu wywołania funkcji za pomocą zegara `CLOCK_HIGHRES` za pomocą programu clockrealtimetiming2.c

Badanie rozdzielczości zegara `CLOCK_REALTIME` za pomocą programu clockrealtimetest.c

Badanie rozdzielczości zegara `CLOCK_HIGHRES` za pomocą programu clockrealtimetest2.c

[4] Przykładowe programy wymienione w tym PDF-ie pochodzą z książki Kay A. Robbins, Steven Robbins „Unix Systems Programming: Communication, Concurrency, and Threads", Prentice-Hall 2003.

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.

Błędy pomiaru czasu

W konstrukcji timerów programowych kryją się mechanizmy generujące błędy pomiaru czasu. Najprostszym błędem czasowym jest **opóźnienie timera** (*timer latency*), które może wynikać po prostu z niezerowego czasu wykonywania pewnych dodatkowych operacji.

Na przykład, jeśli używamy timera cyklicznego z okresem 2 sekund, to po przeterminowaniu się tego timera jest on restartowany, lecz ten restart może potrwać jakiś niewielki okres czasu, np. 5μsekund. W takim przypadku okres cyklu roboczego tego timera wyniesie nie 2 sekundy ale 2.000005. Różnica jest niewielka, ale powstałe opóźnienie będzie się kumulować. Narastający błąd wynikający z kumulowania się błędów pracy okresowej nazywa się **dryfem timera** (*timer drift*).

Łatwo zauważyć, że jeśli timer nie będzie restartowany samoistnie, tylko przez handler sygnału, to pojedyncze opóźnienie takiego restartu może być większe. W tym wypadku nałoży się na nie jeszcze błąd wynikający z rozdzielczości timera. Na przykład, jeśli rozdzielczość timera wynosi 10 ms, a zaprogramowany okres byłby 22 ms (przypadek cokolwiek ekstremalny), to w rzeczywistości obudzenie handlera i restart timera nastąpią po czasie nie krótszym niż 30 ms.

W przypadku pracy cyklicznej można te błędy ograniczyć wykorzystując czas

bezwzględny zamiast względnego. Handler obudzony w każdym cyklu oblicza okres czasu pozostały do następnego nominalnego czasu działania, i programuje dokładnie taki czas. W takim przypadku błędy nie kumulują się, i nigdy nie powinny przekroczyć wartości rozdzielczości timera.

Pracujące cyklicznie timery wykazują jeszcze inny rodzaj błędów określany ogólnie jako **nierównomierności** (*jitter*). Są to niewielkie fluktuacje czasowe, które można podzielić ze względu na ich przyczynę na deterministyczne i losowe. Deterministyczne wynikają z charakterystyki stosowanych algorytmów, lub własności sprzętu obliczeniowego, natomiast losowe wynikają z nakładania się szumów cieplnych i innych zjawisk.

Jednak jest jeszcze jedno zjawisko anormalne możliwe w przypadku pracy cyklicznej timera. Może się zdarzyć, że handler timera obudzi się później niż się spodziewał, lub jego praca przedłuży się aż do wystąpienia kolejnego przeterminowania (handler typowo wstrzymuje lub ignoruje sygnały w czasie swojej pracy). Jeśli wtedy obliczy czas pozostały do tego kolejnego zdarzenia, to może on wyjść ujemny. Zjawisko takie nazywane jest **przekroczeniem timera** (*timer overrun*) i formalnie nie jest błędem, ale może spowodować błąd programu, jeśli nie będzie przewidziane.

Badanie błędów pomiaru czasu programem abstime.c

```
./abstime -a 0.022 1000 0.005
```

Powyższe wywołanie symuluje tysiąc wykonań timera z okresem 22 milisekund, i czasem wykonania handlera sygnału 5 milisekund, wykorzystując zegar czasu absolutnego.

```
./abstime -r 0.022 1000 0.005
```

Powyższe wywołanie powtarza poprzedni eksperyment, ale z wykorzystaniem zegara czasu względnego. Kumulacyjny błąd będzie typowo dużo większy.

```
./abstime -a 0
```

Powyższe wywołanie pozwala oszacować rozdzielczość zegara sprzętowego przez generowanie sygnału po natychmiastowym jednorazowym przeterminowaniu timera.

```
./abstime -a 0.0 1000 0.0
```

Powyższe wywołanie pozwala oszacować maksymalną liczbę sygnałów timera jakie komputer jest w stanie obsłużyć na sekundę wykonując tysiąc przerwań z czasem opóźnienia zero.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Czym różnią się wartości czasowe wykorzystywane przez timery specyfikacji XSI i TMR od interfejsu tradycyjnego?
2. Jakie są operacje możliwe do wykonania na timerze programowym? Jak należy ustawić argumenty wywołania aby wykonać te operacje?
3. W jaki sposób proces może zawiesić swoje wykonywanie na pewien czas?
4. Jakie są możliwe błędy związane z pomiarem czasu?

Przydatne linki

High Resolution Timers, Chapter 5
https://export.writer.zoho.com/public/rreginelli/Chapter-5---High-Resolution-Timers-Final1/fullpage