# Ethernet i AVR–y
# Ethernut od podstaw, część 4


*Oprócz stosu TCP/IP NutOS wyposażono również w podstawowe możliwości systemu operacyjnego czasu rzeczywistego (RTOS). Najważniejszą z nich jest wielowątkowość, czyli równoległe wykonywanie kilku fragmentów programu. Jest ona bardzo przydatna przy tworzeniu aplikacji sieciowych, dlatego czwarty odcinek kursu będzie poświęcony obsłudze wątków w systemie Ethernut.*

Większość przeglądarek internetowych inicjuje jednocześnie kilka połączeń z serwerem. Możliwe jest wówczas równoległe pobieranie np. tekstu (kod HTML) oraz grafiki (pliki GIF, JPG), dzięki czemu strony WWW otwierają się szybciej. Opisany w poprzedniej części kursu serwer WWW mógł obsługiwać w danej chwili tylko jedno połączenie. Wskutek tego, przeglądarka czasami nie wyświetlała obrazka ze strony testowej (logo EP), nie było również możliwe jednoczesne otwarcie strony przez kilku klientów. Nasz serwer powinien więc obsługiwać więcej niż jedno połączenie na raz.

Jednym z rozwiązań tego problemu jest wykorzystanie wielowątkowości (*multithreading* od *thread* – wątek). Pozwala ona na równoległe wykonywanie określonych fragmentów programu, zwanych wątkami. Wielowątkowość można traktować w wielkim uproszczeniu jako prymitywną formę wielozadaniowości (*multitasking*), czyli mechanizmu, dzięki któremu systemy operacyjne mogą wykonywać jednocześnie więcej niż jeden program (zwany w terminologii informatycznej *procesem*). Zasadniczą różnicą między procesami a wątkami jest współdzielenie danych globalnych przez wątki (różne wątki mogą korzystać ze wspólnych zmiennych globalnych). Zmienne lokalne (czyli stos) są oddzielne dla każdego wątku/procesu.

**List. 6. Prosty przykład ilustrujący działanie wątków**

```
THREAD(watek1, args)
{
  int licznik = 0;
  for(;;)
  {
  printf(„Tu watek nr 1 (licznik = %d)!\n”, licznik++);

  NutSleep(500);// czekamy ok. pó? sekundy
  }
}

THREAD(watek2, args)
{
  int licznik = 0;
  for(;;)
  {
  printf(„Tu watek nr 2 (licznik = %d)!\n”, licznik++);
  NutSleep(1000);// czekamy ok. 1 sekundy

  }
}

int main(void)
{
  initialize();

  int i;

  printf(„Kurs EP Ethernut - czesc 4 - prosty przyklad uzywania watkow\n\r\
n\r”);

// tworzymy 2 watki - „Watek1” i „Watek2”
  NutThreadCreate(„Watek1”, watek1, NULL, NUT_THREAD_MAINSTACK);
  NutThreadCreate(„Watek2”, watek2, NULL, NUT_THREAD_MAINSTACK);


// watek „Main” idzie spac
  for(;;)NutSleep(10000);
}
```

## Prosty przykład

Kod prostego programu pokazującego działanie wątków (`Simple-Threads`) znajduje się na **list. 6**. Efektem jego działania jest wysyłanie przez port szeregowy tekstów: `Tu watek nr 1` (co 0,5 sekundy) i `Tu watek nr 2` (co 1 sekundę). Świadczy to o tym, że wątki `watek1` i `watek2` pracują równolegle, ale nie oznacza wcale, że działają one w tym samym czasie. Wykonywanie więcej niż jednego fragmentu kodu w tej samej chwili jest możliwe tylko w systemach wieloprocesorowych lub z procesorami wielordzeniowymi (np. komputery PC z Intel CoreDuo mogą wykonywać w tym samym czasie dwa procesy lub wątki).

W praktyce „jednoczesne” wykonywanie się wątków jest realizowane przez szybkie przełączanie się procesora między nimi. Przełączanie wątków polega na:

– przerwaniu obecnie pracującego wątku i zapisaniu jego stanu w strukturze nazywanej w języku angielskim *switchframe*. Każdy wątek ma własną strukturę *switchframe*, zawiera ona wszystkie parametry procesora (zawartości rejestrów, licznik rozkazów, wskaźnik stosu, wartości flag)

jednoznacznie określające jego stan.
– załadowaniu do rejestrów procesora wartości ze struktury *switchframe* wątku, na który się przełączamy.

Można tu zauważyć podobieństwo do obsługi przerwań – gdy wystąpi przerwanie, stan procesora jest zapisywany na stosie, a po zakończeniu jego obsługi do rejestrów ładowane są uprzednio zapisane wartości.

## Realizacja wielowątkowości w systemie Nut/OS

Za przełączanie wątków w systemie Nut/OS odpowiada funkcja `NutThreadSwitch()`. Jej kod (dla architektury AVR) wraz z odpowiednią strukturą *switchframe* można znaleźć w pliku `arch/avr/os/context_gcc.c`. Funkcja ta nie jest wykorzystywana bezpośrednio w aplikacjach, ale warto poznać zasadę jej działania.

Podział czasu procesora pomiędzy poszczególnymi wątkami w programie z list. 6 przedstawiono na **rys. 4**. Oprócz utworzonych przez nas wątków `watek1` i `watek2` działają także zainicjowane przez system wątki `main` (funkcja `main()`) oraz `idle` (wątek „jałowy" pracujący, gdy pozostałe wątki są w stanie oczekiwania), a jeżeli korzystamy z funkcji sieciowych – jeszcze kilka wątków obsługujących stos TCP/IP.

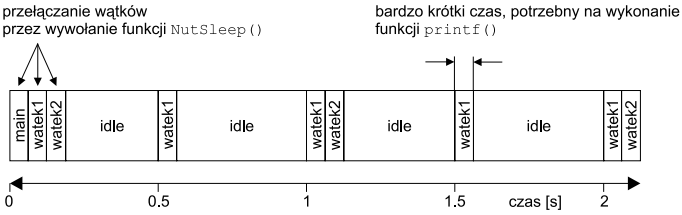
O kolejności wykonywania poszczególnych wątków przez procesor decyduje element systemu operacyjnego nazywany *planistą* (*scheduler*). Za chwilę pokażę, jak działa algorytm planisty w systemie Nut/OS. Ale najpierw trochę definicji: po pierwsze, każdy z wątków w Ethernucie może znajdować się w jednym z niżej wymienionych stanów:
– `TDS_RUNNING` – obecnie pracujący wątek,
– `TDS_SLEEP` – wątek oczekujący na wystąpienie zdarzenia, zakończenie operacji wejścia–wyjścia lub uśpiony na pewien czas przez funkcję `NutSleep()`. Po zakończeniu okresu uśpienia/oczekiwania wątek przechodzi w stan `TDS_READY`,
– `TDS_READY` – wątek gotowy do wznowienia pracy.

Po drugie, każdy wątek ma przypisany priorytet. Jest to liczba z zakresu od 0 do 254, przy czym wbrew logice: 0 oznacza najwyższy, a 254 – najniższy priorytet. Priorytet obecnie pracującego wątku można ustawić za pomocą funkcji `NutThreadSetPriority()`. Najwyższe priorytety (od 0 do 31) są zarezerwowane dla wątków systemu operacyjnego i nie powinno się ich ustawiać. Nowo utworzone wątki mają priorytet równy 64.

Mechanizm działania planisty w systemie Nut/OS jest następujący:
– jeśli pracujący w danej chwili wątek zwolni procesor, system operacyjny przełączy go na

**List. 7. Przykład – animacja tekstu wyświetlanego na LCD zestawu ZL9AVR**

```
// tekst dla wyswietlacza LCD
static char lcd_text[128];

THREAD (lcd_scrolling, args) // wątek zajmujący się animacją tekstu na wyświetlaczu LCD.
{
  int len, x=0, dx = 1, i;

  printf(„Poczatek watku lcd_scrolling!\n\r”);

  for(;;)
  {
  len = strlen(lcd_text);

  if(len<16) // tekst krotszy niz 16 znakow? dopelniamy do 16 spacjami
  {
    for(i=len;i<16;i++) lcd_text[i] = , ,;
    lcd_text[16] = 0;
  }

  if(x>len - 16) { x = 0; dx = 1; }

  LCD_setpos(0);
  LCD_putstringn(lcd_text + x, 16); // 16 - liczba znaków do wyświetlenia na LCD

//przy koncu i poczatku tekstu czekamy troche dluzej
  if(x==0 || x==(len-16)) NutSleep(2000);

  x+=dx;        // przesuwanie tekstu
  if(x==(len-16) || x==0) dx = -dx;

  NutSleep(300);     // czekamy okolo 300 milisekund
  }
}

int CGI_callback(FILE *f, REQUEST *r)  // obsługa skryptu CGI - patrz EP 2/2007
{
(....)
    if(!strcmp(name,”text”))
   {
// parametr text - nowy tekst do wyswietlenia na LCD-ku, kopiujemy go do zmiennej lcd_text
     strcpy(lcd_text, value);
   }
(....)
  return 0;
}

// watek serwera HTTP - obsluga sterowania aplikacja przez przegladarke WWW
THREAD(http_server, args)
{
  TCPSOCKET *s;
  FILE *f;

  printf(„Poczatek watku http_server!\n\r”);

  for (;;) // pętla serwera HTTP - patrz EP 1, 2/2007.
  {
  (...)
  }
}

int main(void)
{
  (...) //inicjalizacja

// tworzymy watek serwera WWW
  NutThreadCreate(„httpd”, http_server, NULL, NUT_THREAD_MAINSTACK);

// tworzymy watek obslugujacy wyswietlacz LCD
  NutThreadCreate(„lcd”, lcd_scrolling, NULL, NUT_THREAD_MAINSTACK);

// ustawiamy najnizszy priorytet watku main
  NutThreadSetPriority(254);

// watek main idzie spac...
  for(;;) NutSleep(10000);

  return 0;
}
```




**Rys. 4. Podział czasu procesora między poszczególnymi wątkami w programie z list. 6**

wątek znajdujący się w stanie `TDS_READY` o najwyższym priorytecie,
– jeśli żaden z wątków nie jest gotowy (stan `TDS_SLEEP`), system przełączy się na wątek `idle` o najniższym priorytecie.

Zastosowanie takiego algorytmu ma jedną bardzo poważną konsekwencję – przełączenie wątków jest możliwe tylko gdy obecnie wykonujący się wątek dobrowolnie zwolni procesor. Możemy to zrobić przez wywołanie jednej z następujących funkcji (dokładny opis w ramce):

– `NutSleep()`,
– `NutThreadYield()`,
– `NutEventWait()`,
– `NutTcpConnect()`, `NutTcpAccept()`, `NutTcpReceive()`, `NutTcpSend()` lub innej blokującej operacji wejścia–wyjścia na gnieździe sieciowym.

Jeżeli o tym zapomnimy, doprowadzimy do zawieszenia się programu – aby się o tym przekonać, możemy usunąć wywołanie funkcji `NutSleep()` w kodzie jednego z wątków (**list. 6**). Wówczas pracować będzie tylko ten wątek, z którego usunęliśmy `NutSleep()`.

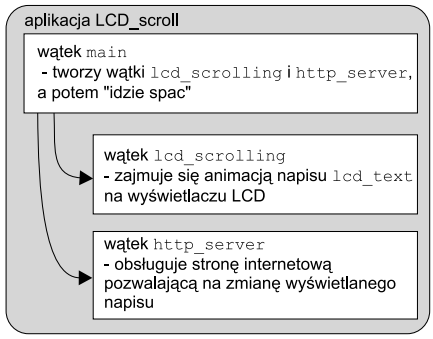
Taka implementacja wielowątkowości jest nazywana kooperatywną (*cooperative multithreading*), poza systemem Nut/OS można ją spotkać w starych Windowsach (wersja 3.x) oraz MacOS do wersji 9. W nowszych systemach stosuje się wielowątkowość (wielozadaniowość) z wywłaszczaniem (*preemptive multitasking*) – wówczas system operacyjny może przerwać aktualnie pracujący wątek (proces) w dowolnej chwili, aby umożliwić pracę innemu. Dzięki temu, system może działać stabilniej (zawieszenie się jednego procesu nie powoduje „zwisu" całego systemu), ale ceną za to jest mniejsza wydajność, bardziej skomplikowane jądro systemu oraz konieczność częstego stosowania skomplikowanych mechanizmów synchronizacji wątków (muteksy, sekcje krytyczne).

## Zarządzanie pamięcią w Nut/OS

Interfejs programistyczny (API) systemu Nut/OS udostępnia standardowe funkcje języka C umożliwiające dynamiczną alokację pamięci – `malloc` oraz `free`. Można je używać w taki sam sposób jak w aplikacjach na komputery PC. Należy przy tym pamiętać, że mamy do dyspozycji niecałe 32 kB pamięci RAM. Ilość pamięci dostępnej w danej chwili zwraca funkcja `NutHeapAvailable`.

## Wątki w praktyce

Wątki są szczególnie przydatne, gdy budujemy urządzenie, które ma wykonywać pewną cykliczną czynność (np. monitorowanie wejść) i jednocześnie mieć możliwość obsługi przez sieć Ethernet. Ilustruje to przykład z **list. 7** (`LCD_scroll`). Jest to program, który wyświetla na wyświetlaczu LCD zestawu ZL9AVR animowany tekst, o treści ustawianej za pomocą przeglądarki internetowej. Program tworzy dwa wątki (patrz **rys. 5**): `lcd_scrolling` zajmujący się animacją napisu oraz `http_server` – serwer WWW (taki sam jak w poprzednim odcinku kursu) obsługujący stronę umożliwiającą zmianę wyświetlanego tekstu.




**Rys. 5. Wykorzystanie wątków w programie z list. 7**

**Ważniejsze funkcje używane w przykładach**

**`THREAD(nazwa_funkcji, nazwa_parametru)`**
Makro generuje deklarację funkcji (wraz z odpowiednimi atrybutami dla kompilatora) mającej pracować jako wątek. Na przykład wywołanie:
THREAD(moj_watek, argumenty) { kod funkcji }
utworzy funkcję o postaci:
// atrybut noreturn jest podpowiedzia dla kompilatora C, ze funkcja nigdy sie nie konczy
void moj_watek(void *argumenty) __attribute__ ((noreturn));
void moj_watek(void *argumenty) { kod funkcji }

**`HANDLE NutThreadCreate(char *name, void (*fn) (void *), void *arg, size_t stackSize);`**
Tworzy nowy wątek o nazwie **`name`**. Wątek zaczyna swoją pracę od wywołania funkcji **`fn`** z parametrem **`arg`**. Przez **`arg`** można przekazać funkcji **`fn`** dowolny wskaźnik jako argument (lub podać NULL, jeżeli go nie potrzebujemy). Wartość **`stackSize`** określa rozmiar stosu dla tworzonego wątku. Domyślna wartość to 768 bajtów (makrodefinicja **`NUT_THREAD_MAINSTACK`**). Nowo utworzony wątek ma priorytet 64.

**`u_char NutThreadSetPriority(u_char level);`**
Ustawia priorytet obecnie pracującego wątku na wartość **`level`**. Należy pamiętać o tym, że wątki tworzone przez użytkownika mogą mieć priorytety z przedziału 32…254, przy czym mniejsza wartość oznacza większy priorytet. Funkcja zwraca wartość priorytetu wątku przed zmianą.

**`void NutThreadYield(void);`**
Wywołanie tej funkcji powoduje natychmiastowe przełączenie bieżącego wątku zgodnie z algorytmem planisty systemu Nut/OS.

**`void NutSleep(u_long ms);`**
Przerywa działający wątek na czas co najmniej **`ms`** milisekund i przełącza go zgodnie z algorytmem planisty. Rzeczywisty czas, na jaki wątek zostanie uśpiony zależy od rozdzielczości timera systemowego (domyślnie 62,5 milisekundy). Funkcji `NutSleep` nie wolno używać do odmierzania precyzyjnych opóźnień!

**`void NutDelay(u_long ms);`**
Czeka dokładnie **`ms`** milisekund. Bieżący wątek nie jest przełączany. Funkcję można stosować do precyzyjnych opóźnień.

**`void *malloc(size_t), free(void *);`**
Funkcje alokujące i zwalniające pamięć zgodne ze standardową biblioteką języka C.

**`void *NutHeapAlloc(size_t), NutHeapFree(void *);`**
Ethernutowe odpowiedniki funkcji `malloc()` i `free()`. Można używać zamiennie z `malloc()` i `free()`.

**`size_t NutHeapAvailable();`**
Zwraca ilość dostępnej pamięci sterty w bajtach.

## Wielowątkowy serwer WWW

Po stosunkowo długim wykładzie na temat wątków, pokażę wspomniany we wstępie ulepszony serwer WWW ze styczniowego odcinka kursu (`HTTPServerMT`), umożliwiający monitorowanie stanu przycisków oraz sterowanie wyświetlaczem i diodami LED w zestawie ZL9AVR. Istotne fragmenty jego kodu przedstawiono na **list. 8.** Program tworzy wątek obsługujący animację napisu na LCD (taki jak w poprzednim przykładzie) oraz pięć identycznych wątków, rozpoczynających swoją pracę od wywołania funkcji `httpd_thread`. Każdy z nich wykonuje opisany w poprzednich odcinkach kursu kod serwera WWW. Jedyną różnicą jest sprawdzanie ilości wolnej pamięci przed wywołaniem funkcji `NutHttpProcessRequest` – funkcja ta wymaga co najmniej 8 kB wolnego RAM–u. Dzięki równoległej pracy kilku wątków serwera możliwa jest jednoczesna obsługa wielu połączeń.

**List. 8. Najistotniejsze fragmenty kodu wielowątkowego serwera WWW**

```
// watek serwera HTTP. Uruchamiamy 5 takich na raz ;)
THREAD(httpd_thread, arg)
{

   for (;;) // petla serwera WWW
  {

  (...)// oczekiwanie na polaczenie (NutTcpAccept)
  f = _fdopen((int) s, „r+b”);

// Sprawdzamy, czy mamy dostatecznie duzo dostepnej pamieci aby obsluzyc za-
pytanie HTTP - wymagane jest
// co najmniej 8 kB. Jesli nie - czekamy az pozostale watki zwolnia troche
pamieci
    while(NutHeapAvailable() < 8192)
    {
      printf(„Za malo pamieci... czekam az ktos inny ja zwolni :(\n”);
      NutSleep(1000);
    }

  (...) // obsluga zapytania HTTP
  }

}

int main(void)
{
  char str[10];
  int i;

(...) // inicjalizacja

// tworzymy watek obslugujacy animowany napis na LCD

  NutThreadCreate(„lcds”, lcd_scrolling, NULL, 256);

// tworzymy 5 watkow serwera WWW.
  for(i=0;i<5;i++)
  {
  sprintf(str,”httpd%d”,i);
  NutThreadCreate(str, httpd_thread, NULL, NUT_THREAD_MAINSTACK);
  }

  NutThreadSetPriority(254);// zmniejszamy priorytet watku main
  for(;;) NutSleep(10000); // i idziemy spac

  return 0;
}
```

## Co dalej?

Niestety ramy tego artykułu nie pozwalają na wyjaśnienie wszystkich zagadnień związanych z wielowątkowością i wielozadaniowością. Więcej informacji na temat realizacji wielowątkowości w Nut/OS można znaleźć np. na stronie *http://www.ethernut.de/pdf/entet100.pdf*. Zainteresowanym polecam także lekturę książki A. Silberschatza „Podstawy systemów operacyjnych”.

Za miesiąc wracamy do zagadnień *stricte* sieciowych, czyli Ethernut jako klient protokołu TCP/IP.

**Tomasz Włostowski, EP**
**tomasz.wlostowski@ep.com.pl**

Kody źródłowe przykładowych programów znajdują się na płycie CD–EP oraz na stronach *http://ethernut.ep.com.pl* oraz *http://wlostowski.ep.com.pl*.