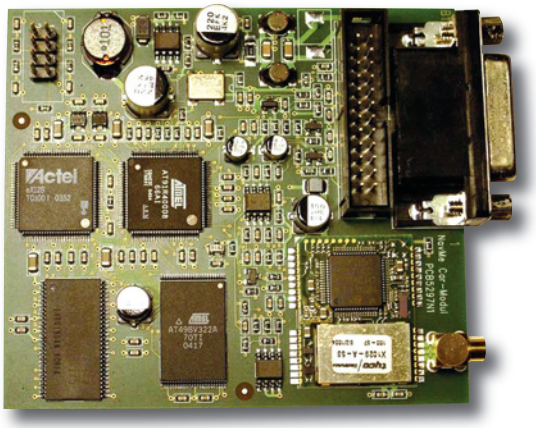
Wprowadzenie do systemów operacyjnych dla systemów wbudowanych na przykładzie platformy ARM, część 3

Kontynuujemy prezentację zagadnień związanych z działaniem systemów operacyjnych, ze szczególnym uwzględnieniem tych, które są stosowane w systemach embedded. Ten cykl jest dobrym przygotowaniem dla programistów zamierzających tworzyć oprogramowanie dla platform tego typu.



Mechanizmy synchronizacyjne

Mamy już w (dużym) przybliżeniu określoną funkcjonalność systemu operacyjnego. Załóżmy też, że mamy oprogramowanie, które potrafi ją zapewnić, czyli stworzyć środowisko pozwalające uruchamianym zadaniom na wykonywanie się quasi–jednoczesne. Dodatkowo zadania o większym priorytecie mogą przerywać wykonywanie tych o mniejszym. Czy to wystarczy do zbudowania naszej aplikacji? Niestety nie. Nawet osoby z małym doświadczeniem w programowaniu *embedded systems* spotkały się z problemem synchronizacji przepływu danych pomiędzy funkcjami obsługi przerwań a resztą kodu. Wystarczy bowiem, że zmienna używana jednocześnie w ISR i w „zwykłym" kodzie nie może być zapisana jedną instrukcją procesora (*non atomic operation*). Na przykład, w 8–bitowym mikrokontrolerze mamy ISR timera, która aktualizuje 32–bitowy licznik milisekund. Jeżeli kod „zwykłej" funkcji chce odczytać taki licznik, to potrzebuje kilku instrukcji procesora, aby pobrać zawartość licznika do rejestrów. Załóżmy, że rozpoczęło się ładowanie licznika do rejestrów i zostały pobrane dwa najbardziej znaczące bajty. W tym momencie następuje przerwanie timera. ISR aktualizuje licznik i kończy działanie, a „normalny" kod kontynuuje pobieranie dwóch najmniej znaczących bajtów licznika. Jeżeli ISR zaktualizowała tylko dwa najmłodsze bajty, to nic złego się nie dzieje, jednakże nie możemy zakładać, że zawsze nastąpi taki korzystny dla nas zbieg okoliczności. W mniej sprzyjających warunkach kończymy odczyt licznika, w którym dwa młodsze bajty są poprawne, a dwa najbardziej znaczące już nie. Pojawia się w tej sytuacji *Duży Kłopot*. Co więcej, bywa też tak, że ograniczenie się przy odczycie/zapisie pamięci do pojedynczej instrukcji asemblerowej może nie wystarczyć, ponieważ przerwanie może zawiesić wykonywanie niektórych pojedynczych rozkazów mikrokontrolera w „połowie drogi". W systemie z wywłaszczaniem (a taki właśnie został powyżej zaprojektowany) trudności te się potęgują – każde zadanie może być zawieszone w dowolnym momencie, w związku z czym inne zadania możemy traktować jak przerwania. Teraz każda zmienna używana równocześnie więcej niż w jednym zadaniu musi być traktowana tak, jakby mogła być zmieniana w ISR. To samo dotyczy dostępu do niektórych zasobów sprzętowych, które wymagają czegoś więcej niż tylko zwykłego odczytu rejestru specjalnego (SFR), np. wykonanie cyklu odczytu z wbudowanego przetwornika A/C.

Najprostszym i jedynym sensownym rozwiązaniem jest blokowanie wszystkich przerwań (w tym zegara systemowego) przed zapisem/odczytem takiej współdzielonej zmiennej. Niestety robiąc to „ręcznie" wracamy do mało elastycznej struktury aplikacji, ponieważ znowu trzeba by dokonać czasochłonnych i precyzyjnych pomiarów tego, co, kiedy i na jak długo może być zablokowane. Nie pozostaje więc nic innego, jak przerzucić na OS–a odpowiedzialność za synchronizację dostępu do zasobów współdzielonych. Zaletą takiego postępowania jest (niejako przy okazji) to, że teraz OS jest w stanie efektywnie kontrolować czas, na jaki są wyłączane przerwania, i co więcej – przesuwać do puli zadań oczekujących (*waiting*) te zadania, które czekają na dostęp do współdzielonych zmiennych/zasobów, a przez to nie „marnować" czasu mikrokontrolera.

Skorzystajmy więc z tego, co już dawno (niektóre z koncepcji mają 40 lat) wymyślono w tej dziedzinie, nie ograniczając się tylko do synchronizacji dostępu do zmiennych.

Zacznijmy od klasyki, czyli mechanizmu semafora. Może on służyć do: kontroli dostępu do zasobu (np. zmiennej), sygnalizowania zajścia jakiegoś zdarzenia i synchronizowania aktywności dwóch zadań. Koncepcja jest dosyć prosta – semafor można sobie wyobrazić jako klucz, który musi przejąć zadanie na własność po to, aby móc dalej kontynuować swoje działanie. Jeżeli semafor jest w danym momencie zablokowany (przejęty) przez inne zadanie, to zadanie chcące go przejąć musi czekać do czasu, kiedy semafor zostanie odblokowany (zwolniony, udostępniony). OS powinien udostępniać co najmniej trzy operacje na semaforze, przy czym każda z nich powinna być wykonywana niepodzielnie (atomowo) – bez możliwości przerwania przez inne zadanie. Są to:

1. Inicjalizuj (*initialize*, *create*) – semaforowi jest przypisywana początkowa wartość liczbowa i tworzona (pusta) lista zadań oczekujących na zwolnienie semafora.

2. Pobierz (*wait*, *pend*) – jeżeli semafor jest dostępny (tj. wartość liczbowa związana z semaforem jest większa niż 0) w momencie wywołania tej funkcji, to zadanie przejmuje semafor (wartość liczbowa semafora jest zmniejszana o 1) i kontynuuje wykonywanie; jeżeli natomiast semafor jest zajęty (wartość semafora jest równa 0), to zadanie wywołujące tę operację jest wciągane na listę zadań oczekujących na zwolnienie semafora i stan zadania jest zmieniany na oczekujący (*waiting*); zwykle możliwe jest wyspecyfikowanie jako parametru funkcji „pobierz" maksymalnego czasu oczekiwania na zwolnienie semafora.
3. Zwolnij (*signal*, *post*) – funkcja ta jest wywoływana przez zadanie, które chce zwolnić semafor; jeżeli nie ma w momencie wywołania funkcji zadań oczekujących (na liście związanej z danym semaforem), to funkcja zwalnia semafor (wartość semafora jest zwiększana o 1) i kończy wywołanie; jeżeli natomiast na liście związanej z semaforem są zadania oczekujące, to jedno z nich jest przesuwane do stanu „gotowe do wykonania" (*ready*), a wartość semafora nie ulega zmianie.

Dwa przykłady użycia semafora pokazano na **list. 4**.

List. 4. Kontrola dostępu do danych

```
typedef struct {
  uint8_t data1;
  int32_t data2;
} ComplexData_T;

ComplexData_T cdata;
// semafor używany do kontroli dostępu do zmiennej  cdata
OSSem *sem;

void init(void)
{
  // utworzenie semafora z początkową wartością 1
  sem = OSSemCreate(1);
}

void task1(void)
{
  uint8_t err;

  ......

  // próbujemy pobrać semafor bez limitu czasu oczekiwania
  OSSemPend(sem, 0, &err);

  // w tym momencie inne zadania będą zawieszane przy wykonywaniu
  OSSemPend(sem)

  cdata.data1 = get_data1();
  cdata.data2 = get_data2();

  // zwalniamy semafor, od tej chwili inne zadania mogą zmieniać lub używać
  // zmodyfikowanych danych w zmiennej cdata
  OSSemPost(sem);

  ......
}

void task2(void)
{
  uint8_t err;

  ......

  // próbujemy pobrać semafor z limitem czasu oczekiwania (10 cykli
  // zegara systemowego)
  OSSemPend(sem, 10, &err);

  // sprawdzamy, czy semafor został prawidłowo przejęty,
  // np. czy nie nastąpiło przekroczenie czasu oczekiwania
  if (OS_OK == err)
  {
    send_data1(cdata.data1);
    send_data2(cdata.data2);
    OSSemPost(sem);
  }

  ......
}
```

Sygnalizowanie zajścia zdarzenia

W tym przypadku semafor jest inicjalizowany wartością 0 (zajęty, zablokowany). Zadanie, które czeka na zajście zdarzenia, wykonuje operację *OSSemPend*, a ponieważ wartość semafora wynosi 0, to zadanie jest zawieszane (przechodzi do stanu *waiting* – nie zajmuje czasu mikrokontrolera) i rozpoczyna oczekiwanie na zajście zdarzenia. Inne zadanie (może to być również procedura obsługi przerwania), które ma do zakomunikowania, że dane zdarzenie się wydarzyło, wykonuje na semaforze operację *OSSemPost* (zwolnienie semafora) i w ten sposób „budzi" zadanie czekające na to zdarzenie. W związku z tym, że oczekiwanie odbywało się w funkcji *OSSemPend*, następuje automatycznie dekrementacja wartości semafora i jego zablokowanie (pod warunkiem, że w międzyczasie nie nastąpiło kolejne zgłoszenie zdarzenia przez *OSSemPost*). Zadanie oczekujące może teraz obsłużyć zdarzenie i cykl może się dalej powtórzyć. Jak widać, taki mechanizm daje się bardzo łatwo rozszerzyć na oczekiwanie i obsługę zdarzeń przez kilka zadań jednocześnie.

Innym popularnym mechanizmem synchronizacyjnym jest kolejka (*queue*). Polega to na tym, że z kolejką jest związana struktura danych pozwalająca na przechowywanie określonej liczby zmiennych (zwykle wskaźników typu **void*), dla których zostają udostępnione atomowe (niepodzielne) operacje dodawania i pobierania tych danych. Jeżeli kolejka w danej chwili jest pusta, to operacja pobierania czeka na nowe dane w ten sam sposób, jak w semaforze – zadanie jest zawieszane (stan *waiting*) i nie „konsumuje" czasu procesora.

Jeszcze inny mechanizm synchronizacyjny to flagi zdarzeń (bity). Pozwalają one w bardzo wygodny sposób zrealizować oczekiwanie na zajście jednego lub kilku zdarzeń. Operacje (jak zwykle atomowe) wykonywane na flagach to: ustawienie/wyzerowanie bitu lub bitów i oczekiwanie na ustawienie/wyzerowanie bitu lub bitów. Dla drugiej z nich można zwykle oprócz maksymalnego czasu oczekiwania podać, czy chcemy czekać, aż wszystkie bity przyjmą pożądany stan, czy też wystarczy tylko jeden z nich. Inną opcją dla operacji oczekiwania jest sposób zachowania po zajściu oczekiwanego zdarzenia, tzn. czy bity zostaną „skonsumowane", czy pozosta-

ną niezmienione. W tym przypadku „konsumpcja" bitów polega na ustawieniu ich w stan przeciwny do tego, na jaki czekaliśmy.

Inne popularne mechanizmy synchronizacyjne to: *mutex*, skrzynka pocztowa (*mailbox*), komunikaty (*messages*), a w systemach wieloprocesorowych – *spinlock*.

Zagadnienie synchronizacji jest bardzo rozległe i nie zostanie ono w tym artykule szerzej omówione. Na koniec warto tylko wspomnieć, że niezwykle istotne jest dobranie właściwego mechanizmu synchronizacyjnego do konkretnego problemu i że synchronizacja w systemach wielozadaniowych (lub wielowątkowych) to źródło największych i zwykle najtrudniejszych do znalezienia błędów projektowych i programistycznych. Jako przykład niech posłuży następująca sytuacja. Wyobraźmy sobie 3 zadania, które mają różne priorytety: zadanie 1 ma najwyższy, 2 mniejszy niż 1, a 3 najniższy. Na początku zadania 1 i 2 czekają na wystąpienie jakiegoś zdarzenia, a zadanie 3 jest w trakcie wykonywania. W trakcie swojego działania zadanie 3 przejmuje semafor potrzebny do uzyskania dostępu do zasobu współdzielonego. Po przejęciu semafora zaczyna ono wykonywać operacje na tymże zasobie. W tym momencie zachodzi zdarzenie, na które czekało zadanie 1 i OS decyduje się na przełączenie do stanu wykonywania zadania 1 (ma ono przecież największy priorytet), zawieszając wykonanie zadania 3. Zadanie 1 wykonuje się bez przeszkód do momentu, gdy próbuje przejąć ten sam semafor, którego używa zadanie 3. Ponieważ semafor jest zablokowany, to zadanie 1 zostaje zatrzymane i oczekuje na zwolnienie się semafora. Zadanie 3 wznawia działanie, ale jest szybko przerywane, ponieważ zaszło zdarzenie, na które właśnie czekało zadanie 2 (o większym priorytecie niż 3). Zadanie 2 zaczyna swoje normalne wykonywanie i obsługę zdarzenia. W tym momencie widać, że efektywnie zadanie o mniejszym priorytecie (zadanie 2) blokuje wykonanie zadania o priorytecie wyższym (zadanie 1). Jest to spowodowane tym, że zadanie 1 czeka na zwolnienie semafora blokowanego przez zadanie 3, które jest z kolei wstrzymywane przez zadanie 2. Jest to typowy problem, który został dobrze opisany w literaturze pod nazwą „odwrócenia priorytetów" (*priority inversion*).

Inne usługi systemu operacyjnego

Zastanówmy się przez chwilę, jakie jeszcze inne ogólne (niezwiązane z konkretną aplikacją) usługi powinien oferować OS.

Ze względu na użyteczność w prostych systemach wbudowanych mogłyby być przydatne co najmniej cztery grupy takich usług.

1. Zarządzanie zadaniami cyklicznymi. Zwykle można to realizować na co najmniej dwa sposoby: poprzez nakazanie cyklicznego uruchamiania zadania, tzn. co określony czas zadanie jest ustawiane w stan „gotowe do wykonania" (*ready*) lub poprzez funkcje zawieszające wykonanie zadania na określony czas.
2. Zarządzanie przydziałem pamięci. W systemach czasu rzeczywistego zwykle nie można bazować na standardowych mechanizmach języka C (*malloc*, *free*) do dynamicznego przydziału pamięci. Z tego względu wymyślono bardziej przewidywalne mechanizmy, spośród których jednym z popularniejszych jest tzw. pula pamięci (*memory pool*) – polega to na tym, że OS podczas inicjalizacji rezerwuje zadaną liczbę kawałków pamięci o określonym rozmiarze, a w trakcie normalnej pracy system (na żądanie) udostępnia zadaniom poszczególne kawałki, które po wykorzystaniu są zwracane do puli.
3. Obsługa funkcji typu *callback* – możliwość automatycznego uruchamiania zadanych funkcji przy zajściu pewnych zdarzeń w systemie, np. wspomniany *idle task* mógłby wywoływać naszą funkcję do zarządzania poborem mocy, lub też każde przełączenie kontekstu zadań mogłoby wywoływać funkcję, która sporządza statystykę działania aplikacji itp.
4. Sterowniki urządzeń. W przeciwieństwie do systemów biurkowych nie każdy (RT)OS przeznaczony do zastosowania w aplikacjach wbudowanych oferuje takie udogodnienia. Wiąże się to głównie z bardzo dużym zróżnicowaniem sprzętu, na jakim takie systemy są uruchamiane.

Artur Lipowski
Cezary Worek