Zarządzanie zadaniami w systemach embedded

Super Simple Tasker, część 1

Dla tych, którzy po raz pierwszy zetknęli się z systemami operacyjnymi typu embedded, jak na przykład µC/OS czy eCos, sposób ich działania może wydać się skomplikowany, sztuczny i mało intuicyjny. W miarę nabierania doświadczenia odczucie to wprawdzie słabnie, nadal pozostaje jednak pewna nieufność. W niniejszym artykule przedstawiamy prostsze rozwiązania alternatywne pozwalające zarządzać zadaniami w systemie embedded.

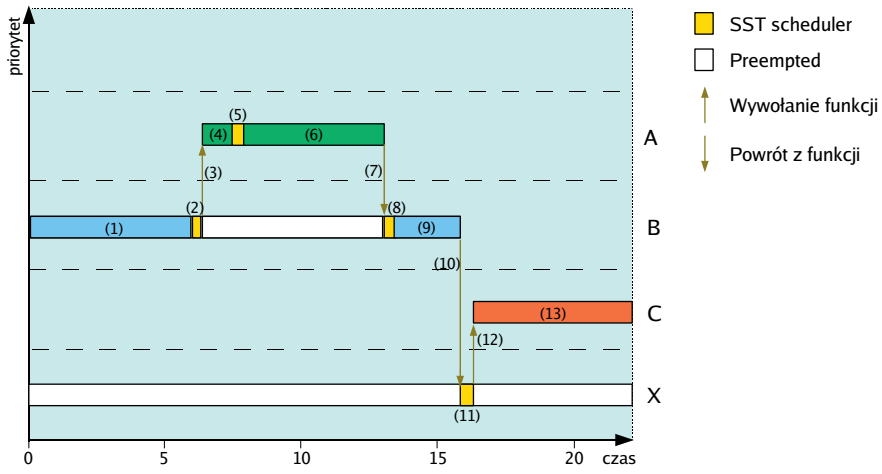
Artykuł jest ilustracją tezy mówiącej o tym, że nawet prosty pomysł może być dobrym rozwiązaniem dla skomplikowanego problemu. Należy zaznaczyć, że przedstawiony pomysł nie pretenduje do bycia zamiennikiem dla powszechnie używanych systemów operacyjnych, a jedynie ma na celu prezentację (kolejnego) narzędzia możliwego do zastosowania w niezbyt rozbudowanych urządzeniach typu *embedded*. W artykule zostanie przedstawiona koncepcja nazywana zwykle „*Super Simple Tasker*" (SST), którą można traktować jako jądro (*kernel*) małego i bardzo prostego systemu operacyjnego.

Budowę zarządcy zadań (*schedulera*) rozpocznijmy od obserwacji dotyczącej typowego cyklu życia zadania (*task*) w systemie osadzonym (*embedded*). Typowy scenariusz rozpoczyna się od oczekiwania na jakieś zdarzenie: nadejście danych, upłynięcie określonego czasu lub np. zmianę stanu pinu I/O. Potem następuje obsługa zdarzenia i zakończenie pracy, czyli *de facto* oczekiwanie na następne zdarzenie. Możliwe są tu oczywiście wszelkie kombinacje związane z przebiegiem zdarzeń, np. jedno zdarzenie (*event*) uruchamia kilka zadań, zdarzenia muszą wystąpić w określonej kolejności, aby zadanie mogło się wykonać itp. Powyższy opis doskonale pasuje także do tzw. maszyny stanów (*Finite State Machine* – FSM), która jest typowym sposobem implementacji funkcji w systemach osadzonych. Z punktu widzenia projektanta systemu byłoby więc najbardziej naturalne, gdyby system po prostu wywoływał funkcję (zadanie) w odpowiedzi na określone zdarzenie, a funkcja ta po wykonaniu swojej pracy zwracałaby (*return*) sterowanie do systemu, czyli zadanie byłoby wykonywane w jednym przebiegu bez nieskończonych pętli (*Run To Completion* – RTC). Jak na razie nie widać w takim założeniu nic specjalnie użytecznego, tym bardziej, że w systemie czasu rzeczywistego (*Real Time* – RT) musi być zapewniony odpowiedni czas reakcji na zdarzenia, a co za tym idzie, należy uwzględnić fakt, że są zadania mniej i bardziej „ważne". Aby tego dokonać, należy w jakiś sposób umożliwić przerywanie jednych zadań przez inne, z uwzględnieniem ich priorytetów. Można to zrobić przez zapamiętanie kontekstu (w tym wszystkich rejestrów procesora) aktualnie wykonywanego zadania i odtworzenie kontekstu zadania, do którego w danej chwili należy się przełączyć, ale w ten sposób wracamy do „klasycznego" OS–a, czego przecież chcemy uniknąć.

Trzeba zatem przyjąć kolejne założenie – przełączenie zadań będzie możliwe tylko drogą „naturalną", a mianowicie przez bezpośrednie wywołanie funkcji *schedulera* (synchronicznie) lub poprzez przerwanie (asynchronicznie). W obu przypadkach kontekst przerywanego zadania jest automatycznie zapamiętywany na stosie i jest to całkowicie (automatycznie) zarządzane przez kompilator. W celu otrzymania pełnej wymaganej funkcjonalności przełączania zadań należy minimalnie rozbudować procedury obsługi przerwań (*Interrupt Service Routine* – ISR). Jeżeli umieści się w ISR wywołanie zarządcy zadań, i to w taki sposób, aby zadania przez niego uruchomione „nie widziały", że są wykonywane z ISR–a, to otrzymamy wielozadaniowy system operacyjny z wywłaszczaniem.

Reasumując, przyjęto następujące założenia:

– zadanie OS–a jest zwykłą funkcją, która wykonuje się i kończy, tzn. nie zawiera w sobie nieskończonych pętli,



Rys. 1. Scenariusz 1

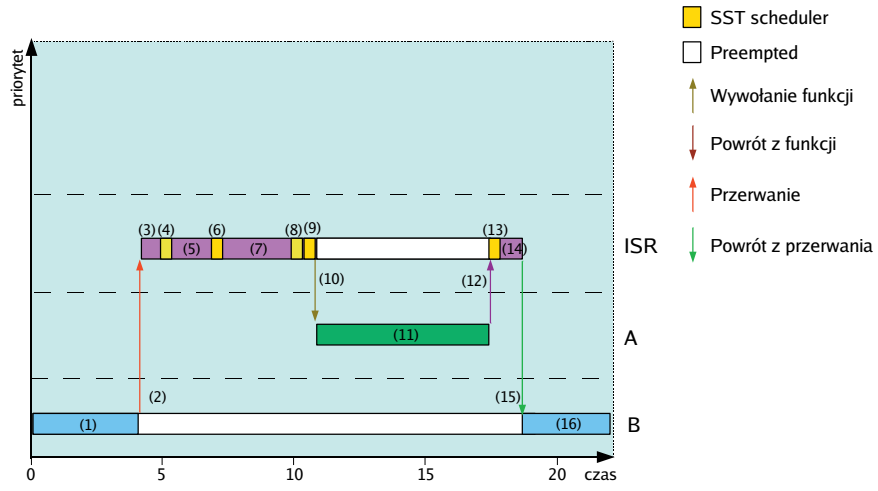
– zadanie może być przerwane (wywłaszczone) tylko przez zadanie o wyższym priorytecie,
– wywłaszczenie jest możliwe tylko w dwóch przypadkach: „dobrowolne" wywołanie *schedulera* (wywłaszczenie synchroniczne) lub przerwanie (wywłaszczenie asynchroniczne),
– podjęcie decyzji, które zadanie będzie wykonywane (*scheduling*), jest możliwe tylko po zakończeniu wykonywania się zadania (funkcji) lub po jego wywłaszczeniu.

W przedstawianym systemie zadania mogą się znajdować tylko w trzech stanach:
– wykonywane (*running*),
– wywłaszczone przez zadanie o wyższym priorytecie (*preempted*),
– gotowe do wykonania (*ready).*

Czytelnicy dobrze znający tradycyjne systemy operacyjne dostrzegą, że brakuje tu charakterystycznego dla nich stanu oczekiwania (*waiting*).

Z przyjętych założeń wynikają bardzo istotne (dla projektanta) implikacje:

1. Zadanie nie jest i nie może być zawieszane (*suspended*) w oczekiwaniu na zdarzenie zewnętrzne. Jest to największa wada i jednocześnie zaleta opisywanego rozwiązania. Wada, gdyż większość osób piszących programy jest przyzwyczajona do wywoływania funkcji, które zatrzymują pracę zadania/wątku, aż do otrzymania konkretnego zdarzenia (np. funkcja *getchar*). Zaleta, gdyż upraszcza w sposób zasadniczy budowę całego systemu operacyjnego, a właściwie zarządcy zadań.
2. Żadne nowe zadanie nie może zacząć się wykonywać, dopóki wszystkie zadania (w stanie gotowe do wykonania lub przerwane (*preempted*)) o takim samym lub wyższym priorytecie nie zostaną zakończone. Zadanie w trakcie swojego wykonywania skutecznie blokuje wszystkie inne zadania o takim samym lub wyższym priorytecie.
3. Z punktu 2 wynika bezpośrednio, że zadania o tym samym priorytecie są serializowane i współdzielenie zasobów pomiędzy nimi nie wymaga synchronizacji. W przypadku, gdy zadania o różnych priorytetach używają tego samego zasobu (np. struktury danych), staje się niezbędna synchronizacja.
4. Zadanie jest normalną funkcją, dlatego czas życia zadań i ich lokalnych zmiennych (przechowywanych na stosie) jest taki sam. W momencie zakończenia zadania znikają jego dane. Jeśli zadanie wymaga przechowania informacji pomiędzy swoimi wywołaniami, musi być to zrobione przez zmienne globalne. Stanowi to istotną wadę w dużych systemach (brak skalowalności).



Rys. 2. Scenariusz 2

Po tych nieco teoretycznych wywodach, pora na demonstrację „jak to działa?". Na początek przyjmijmy, że w systemie są uruchomione tylko trzy zadania: A, B i C. Zadanie A ma najwyższy priorytet, a zadanie C najniższy. Załóżmy, że pierwszy scenariusz (**rys. 1**) rozpoczyna się w momencie, kiedy zadanie B jest właśnie wykonywane i w tym czasie następuje generacja zdarzenia przeznaczonego dla zadania A. Jaki w tym przypadku będzie przebieg zdarzeń? Zadanie B stwierdza w trakcie przetwarzania danych (1), że ma do przekazania informację (komunikat) dla zadania A. Aby to zrobić, wywołuje funkcję OS–a służącą do wysyłania komunikatów. Uruchomienie tej funkcji powoduje umieszczenie komunikatu w odpowiedniej kolejce i przeniesienie zadania A do stanu gotowe do wykonania (z powodu komunikatu oczekującego na obsłużenie), a następnie wywołanie (funkcji) zarządcy zadań (2). Zarządca analizuje listę zadań będących w stanie gotowe do wykonania i stwierdza, że zadanie A ma aktualnie najwyższy priorytet, po czym zaczyna je wykonywać (3), czyli wywołuje (jego) funkcję. Zadanie A zaczyna przetwarzanie (4), w trakcie którego wysyła komunikat do zadania C. Zadanie C przechodzi do stanu gotowe do wykonania (5), ale ze względu na to, że priorytet C jest mniejszy niż (aktualnie wykonywanego) A, nie następuje uruchomienie zadania C. Po pewnym czasie (6) zadanie A kończy się (7), a *scheduler* wznawia swoją pracę (8), tzn. ponownie przegląda listę wszystkich zadań gotowych do wykonania. Teraz jest tam tylko zadanie C z priorytetem mniejszym niż (przerwane B), B może zatem kontynuować (9), a następnie zakończyć (10) swoją pracę. Dopiero po jego zakończeniu znowu jest wywoływany zarządca zadań (11). Jeżeli na liście zadań gotowych do wykonania nie ma żadnego zadania z priorytetem wyższym niż C, wówczas to właśnie zadanie może być (w końcu) uruchomione (12), (13).

W kolejnym scenariuszu (**rys. 2**) będzie opisany ciekawszy przypadek, tj. wystąpienie przerwania. Tak jak w poprzednim przykładzie, opis rozpoczyna się w trakcie wykonywania zadania B (1). Zakładamy także, iż przerwania są odblokowane.

W momencie wystąpienia przerwania (2) procesor przestaje wykonywać zadanie B i automatycznie rozpoczyna wykonywanie kodu odpowiedniej procedury obsługi przerwania. W ISR po wykonaniu operacji specyficznych dla źródła przerwania (przykładowo zablokowanie przerwań spowodowanych tą

List. 1. Podniesienie priorytetu wykonywanego zadania

```
SST_Mutex_T SST_MutexLock(CPU_Base_T prioCeiling)
{
    SST_DECLARE_INT_USAGE;
    SST_Mutex_T p;
    SST_INT_LOCK_AND_SAVE(); (1)
    p = _SST_CurrPrio; (2)
    if (prioCeiling > _SST_CurrPrio) (3)
    {
      _SST_CurrPrio = prioCeiling;   (4)
    }
    SST_INT_LOCK_RESTORE();  (5)
    return p;
}

void SST_MutexUnlock(SST_Mutex_T mutex)
{
    SST_DECLARE_INT_USAGE;
    SST_INT_LOCK_AND_SAVE(); (6)
    if (mutex < _SST_CurrPrio)  (7)
    {
      _SST_CurrPrio = mutex;    (8)
      _SST_Schedule(SST_INT_STATE);  (9)
    }
    SST_INT_LOCK_RESTORE();  (10)
}

Blokowanie (SST_MutexLock) i zwalnianie (SST_MutexUnlock) zasobu:
 (1) - Zablokuj przerwania.
 (2) - Zapamiętaj priorytet aktualnie wykonywanego zadania.
 (3) - Czy priorytet docelowy jest większy od aktualnego?
 (4) - Jeżeli tak, to zmień priorytet wykonywanego zadania na docelowy.
 (5) - Odblokuj przerwania.
 (6) - Zablokuj przerwania.
 (7) - Czy priorytet zapamiętany jest mniejszy od aktualnego?
 (8) - Jeżeli tak, to zmień priorytet wykonywanego zadania na zapamiętany.
 (9) - Wywołaj scheduler.
(10) - Odblokuj przerwania.
```

List. 2. Sposób użycia *mutex–a*

```
static void print(const char *str)
{
    SST_Mutex_T mutex;
    mutex = SST_MutexLock(TIMER_TASK_PRIO_CEIL);
    while (*str)
    {
      SST_Wait_Busy_While(IS_BUF_FULL(tx_buf));
      put_buf(&tx_buf, *str);
      str++;
    }
    SST_MutexUnlock(mutex);
}
```

sama przyczyną, np. nieodebranym bajtem danych w buforze UART–a) (3) uruchamiany jest kod (*SST ISR entry*) (4) specyficzny dla naszego systemu. Zawiera on zapamiętanie priorytetu zadania wykonywanego w momencie wystąpienia przerwania, ustawienie aktualnego priorytetu na wartość odpowiadającą przerwaniu (jest to priorytet wyższy od priorytetu któregokolwiek z zadań w systemie) i opcjonalnie odblokowanie przerwań. Następnie jest wykonywane przetwarzanie specyficzne dla danego ISR–a (5), w tym wysłanie komunikatu (np. informacji o właśnie odczytanej wartości przetwornika AD) do zadania A. W trakcie wysyłania komunikatu zadanie A jest ustawiane w trybie gotowe do wykonania (6), ale że aktualny priorytet jest bardzo wysoki (przerwanie), zadanie to nie jest od razu wykonywane. Dalej wykonywanie kodu specyficznego dla danego przerwania zostaje zakończone (7) i następuje powtórne przejście do kodu naszego systemu (*SST ISR exit*) (8). W tym momencie przerwania są blokowane (jeżeli były odblokowane w kodzie *ISR entry*), wysyłany jest sygnał do kontrolera przerwań (jeżeli taki występuje) o zakończeniu przerwania, zostaje odtworzony priorytet zadania przerwanego przez przerwanie (w tym przypadku priorytet B) i na koniec wywoływany jest *scheduler* (9). Po uruchomieniu zarządca zadań stwierdza, że w stanie gotowe do wykonania jest zadanie A z większym priorytetem od aktualnego zadania (B). Oczywiście powoduje to, że zadanie A zostaje wykonane (10), (11), (12) (jest to zwykłe wywołanie funkcji języka C). W związku z brakiem innych zadań gotowych do wykonania, *scheduler* kończy swoją pracę (13), po czym jest wykonywana reszta kodu ISR (14), czyli powrót z przerwania (15). W ten sposób zostaje wznowione wykonanie zadania B (16).

Zadanie A jest wykonywane z odblokowanymi przerwaniami, dlatego cały scenariusz może być powtarzany w sposób rekurencyjny i to włącznie z wystąpieniem i obsługą tego samego zdarzenia, które zapoczątkowało rozważany przypadek. Na pierwszy rzut oka może się to wydawać pogwałceniem zasady, zgodnie z którą czas wykonywania procedury obsługi przerwania powinien być jak najkrótszy. Wrażenie to bierze się stąd, iż nie można czasu trwania ISR–a definiować jako okresu pomiędzy wystąpieniem przerwania i zapamiętaniem kontekstu przerwanego zadania a odtworzeniem kontekstu i powrotem po wykonaniu instrukcji „*IRET*”. W przypadku SST należy założyć, że przerwanie kończy się w momencie wykonania kodu *SST ISR exit*, po czym system przestaje pracować w kontekście (trybie) przerwania.

Synchronizacja dostępu

Jak już wspomniano, zadania z takim samym priorytetem są automatycznie serializowane i mogą współdzielić zasoby bez potrzeby synchronizowania dostępu. Jeżeli jednak zasób jest wykorzystywany przez zadania o różnym priorytecie, niezbędne staje się użycie mechanizmów synchronizujących. W przypadku SST powinien to być bardzo prosty mechanizm, aby nie skomplikować całości rozwiązania. Wydaje się, że najprostszym możliwym rozwiązaniem jest *mutex*, korzystający z właściwości SST. Wystarczy wziąć pod uwagę cechę SST polegającą na tym, że nowe zadanie może się rozpocząć (przejść do stanu wykonywane) tylko wtedy, gdy wszystkie inne zdania o takim samym lub wyższym priorytecie się skończyły. Jeżeli zatem na czas dostępu do chronionego zasobu dałoby się podnieść priorytet wykonywanego zadania, np. do wartości N, wówczas żadne inne zadanie o priorytecie mniejszym lub równym N nie miałoby możliwości wykonywania się w tym samym czasie, a przez to nie miałoby także dostępu do chronionego zasobu. Kod takiego rozwiązania może wyglądać jak na **list. 1**, a sposób użycia przedstawiono na **list. 2**.

Artur Lipowski
LAL@pro.onet.pl