Zastosowania procesorów sygnałowych

SYSTEMY OPERACYJNE PROCESORÓW SYGNAŁOWYCH

Opracowanie: Grzegorz Szwoch

Politechnika Gdańska, Katedra Systemów Multimedialnych

Potrzeba programowania wielowątkowego

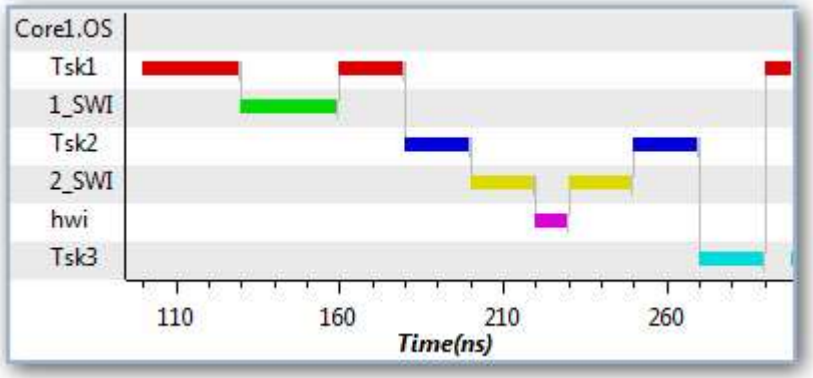
- Podczas laboratorium ZPS piszemy proste programy, które są uruchamiane sekwencyjnie, instrukcja po instrukcji, na procesorze sygnałowym.
- Taki sposób programowania nazywa się *bare metal* - bezpośrednio na procesorze.
- W ten sposób można tworzyć tylko bardzo proste programy.
- Jeżeli program oczekuje na dane wejściowe (np. z czujnika), nie może wykonywać w tym czasie innych operacji – marnujemy cykle procesora (czas bezczynności).
- Z drugiej strony, jeżeli przetwarzanie danych trwa zbyt długo, możemy utracić dane wejściowe.
- Z tych powodów, bardziej złożone programy na procesorach sygnałowych tworzone są jako programy wielowątkowe.

Operacje wejścia/wyjścia

- Operacje wejścia/wyjścia (*input/output* I/O): pobieranie danych wejściowych, wysyłanie wyników na wyjście.
- Powodują zawieszenie wykonywania programu – czekanie na dostępność danych.
- Problematyczne np. gdy program pobiera dane z kilku czujników.
- Chcemy wykonywać obliczenia w czasie, gdy czekamy na nowe dane.
- Rozwiązanie: podzielenie programu na wątki.
- Każdy wątek wykonuje wydzieloną część programu. Np. jeden odbiera dane z czujnika, drugi wykonuje analizę danych, trzeci wysyła wyniki, itp.
- Wątki mogą być wstrzymane i wznowione w dowolnej chwili.
- Optymalizacja wykorzystania dostępnych zasobów procesora sygnałowego.

Programowanie wielowątkowe

- Proces – działający program, uruchomiony na procesorze.
- Wątek (*thread*) lub zadanie (*task*) – wyodrębniona część programu.
- W ramach procesu może działać jeden lub więcej wątków.
- Jeżeli w procesie jest więcej niż jeden wątek, nazywamy to programowaniem wielowątkowym (*multithtreading*) lub współbieżnym (*concurrency*).
- Wątki rywalizują o dostęp do zasobów (*resources*): cykli procesora, pamięci.
- Można uruchomić wiele wątków w ramach programu.
- W danej chwili, instrukcje mogą być wykonywane na jednym rdzeniu procesora przez tylko jeden wątek.
- Pozostałe wątki są w tym czasie wstrzymane (uśpione).



Współbieżność a zrównoleglenie

Współbieżność (*concurrency*):

- wiele wątków może być uruchomionych jednocześnie,
- jeden wątek otrzymuje dostęp do rdzenia procesora, pozostałe są wstrzymane,
- większość klasycznych procesorów sygnałowych jest jednordzeniowa.

Zrównoleglenie (*parallelism*):

- procesor posiada wiele jednostek obliczeniowych („rdzeni”),
- kilka wątków jednocześnie wykonuje operacje na procesorze,
- zrównoleglenie SIMD: wszystkie wątki wykonują te same operacje,
  ale na różnych danych wejściowych (stosowane np. na GPU),
- wymaga wielordzeniowego procesora sygnałowego lub układu wielu procesorów,
- procesory ogólnego przeznaczenia, np. ARM, są często wielordzeniowe.

Typy wielozadaniowości

Wielozadaniowość współpracująca (*cooperative multitasking*):

- wątek sam decyduje kiedy „zwolnić procesor”,
- stosowany gdy programista ma pełną kontrolę nad działaniem wątków,
- istnieje ryzyko, że jeden wątek „zawiesi” cały program.

Wielozadaniowość z wywłaszczaniem (*preemptive multitasking*):

- istnieje nadrzędny wątek – system operacyjny, który zarządza pracą innych wątków,
- każdy wątek może w dowolnym momencie zostać zatrzymany (wywłaszczony) na rzecz innego wątku,
- bardziej wydajny pod kątem wykorzystania zasobów, więcej możliwości,
- stosowane na procesorach sygnałowych w większości nietrywialnych programów.

Przełączanie kontekstu

- Kontekst (*context*) – uruchomiony wątek i przydzielone mu zasoby (cykle procesora).
- Przełączenie kontekstu (*context switch*) – zawieszenie działania wątku i przekazanie zasobów innemu wątkowi.
- Priorytet (*priority*) – liczba określająca hierarchię (ważność) wątków.
- Przełączenie kontekstu następuje najczęściej w dwóch przypadkach:
  - wątek zasypia (*sleep*), np. czekając na dane („dobrowolne” oddanie zasobów),
  - wątek o wyższym priorytecie niż aktualnie działający żąda dostępu do zasobów (wywłaszczenie wątku i przejęcie zasobów).
- Program z wielozadaniowością współpracującą umożliwia tylko pierwszy przypadek. Drugi wymaga wielozadaniowości z wywłaszczaniem – system operacyjny musi zarządzać dostępem wątków do zasobów.

Konflikty między wątkami

- W programie wielowątkowym mogą występować sytuacje konfliktowe.
- Przykład:
  - wątek B przetwarza wartości zapisane w tablicy,
  - w trakcie tej operacji, wątek A o wyższym priorytecie przejmuje kontekst i zapisuje nowe wartości do tej samej tablicy,
  - wątek B otrzymuje z powrotem kontekst i kontynuuje przetwarzanie tablicy,
  - efekt: wątek B przetwarza „miks” starych i nowych wartości z tablicy,
  - może się nawet zdarzyć, że jedna liczba zostanie nadpisana częściowo (np. liczba typu *short*: jeden bajt ze „starej” liczby, drugi z „nowej”).
- Takie sytuacje skutkują błędnymi wynikami obliczeń. Trzeba zablokować możliwość zmodyfikowania danych, na których pracuje wątek.

Muteks

- Muteks (*mutex*) – obiekt pozwalający na wyłączny dostęp wątku do zasobu.
- Inne nazwy: blokada (*lock*), bramka (*gate*).
- Wątek, który chce uzyskać dostęp do zasobu, musi pobrać i zablokować muteks.
- Muteks należy zwolnić po wykonaniu operacji.
- Jeśli muteks jest zablokowany przez jeden wątek, drugi wątek czeka na jego zwolnienie, lub rezygnuje i wykonuje w tym czasie inne operacje.
- Sekcja krytyczna (*critical section*) – fragment kodu zabezpieczony muteksem, który powinien być wykonany w całości (bez przerwy) przez jeden wątek.
- Stosowanie muteksów spowalnia program. Należy je stosować tam, gdzie są konieczne i nie blokować muteksów dłużej, niż to potrzebne.

Przykład zastosowania muteksu

- Wątek B (niższy priorytet) blokuje wolny muteks i rozpoczyna przetwarzanie danych zapisanych w tablicy.
- W trakcie tej operacji, wątek A (wyższy priorytet) przejmuje kontekst i próbuje zablokować muteks do zapisu nowych danych do tablicy.
- Ponieważ muteks jest zablokowany, wątek A czeka.
- Wątek B odzyskuje kontekst, nadal blokuje muteks, więc może kontynuować przetwarzanie danych z tablicy.
- Po zakończeniu przetwarzania, wątek B zwalnia muteks.
- Wątek A otrzymuje odblokowany muteks. Blokuje go, zapisuje nowe dane do tablicy i zwalnia muteks.
- Wątek B może przejąć muteks i kontynuować cały cykl.

Przykład zastosowania muteksu

```
// Wątek A – wyższy priorytet

mutex.lock();  // próba zablokowania

// tutaj przechodzimy dopiero
// gdy zablokowanie się powiedzie
int i;
for (i = 0; i < 512; i++) {
    tablica[i] = pobierz_dane();
}

mutex.unlock();  // zwolnienie
```

```
// Wątek B – niższy priorytet

mutex.lock();  // próba zablokowania

// tutaj przechodzimy dopiero
// gdy zablokowanie się powiedzie
int i;
for (i = 0; i < 512; i++) {
    przetworz_dane(tablica[i]);
}

mutex.unlock();  // zwolnienie
```

Semafor

- Semafor (*semaphore*) jest obiektem, który umożliwia dostęp wielu wątków do ograniczonej liczby zasobów, np. buforów w pamięci.
- Semafor posiada licznik (*counter*) dostępnych zasobów.
- Pobranie zasobu zmniejsza licznik semafora o 1, zwolnienie: zwiększa o 1.
- Nie można pobrać zasobu, gdy licznik jest równy 0.
- Przykład: jest 5 buforów w pamięci
  - aktualny licznik: 1 (cztery bufory już pobrane),
  - wątek A pobiera bufor, licznik: 0,
  - wątek B nie może już pobrać bufora, musi poczekać,
  - wątek C zwalnia bufor, licznik: 1,
  - wątek B pobiera bufor, na który czekał; licznik: 0.

Kolejki

- Wątki potrzebują przekazywać dane między sobą.
- Typowy schemat przetwarzania – „producent-konsument”:
  - wątek „producent” generuje dane (np. pobiera z czujnika),
  - wątek „konsument” przetwarza te dane.
- Wymiana danych między wątkami wymaga tablicy w pamięci, do której producent zapisuje dane, a konsument czyta z niej dane. Tablica musi być chroniona muteksem.
- Jest to na tyle często stosowany model przetwarzania, że zazwyczaj dostępne są specjalne struktury – kolejki (*queue*), wewnętrznie chronione muteksem (nie trzeba ich blokować i zwalniać ręcznie).
- Producent wstawia dane do kolejki, konsument wyjmuje dane z kolejki.

Kolejki

- Wstawianie (*put*) danych do kolejki:
  - producent próbuje wstawić nowe dane,
  - kolejka jest w tym czasie zablokowana do odczytu,
  - jeżeli kolejka jest pełna (*overflow*)*:* nie można wstawić nowych danych, wtedy wątek może czekać na zwolnienie miejsca (blokowanie) lub zrezygnować i przejść do wykonywania innych operacji.
- Wyjmowanie (*get*) danych z kolejki:
  - konsument próbuje odczytać nowe dane,
  - kolejka jest w tym czasie zablokowana do zapisu,
  - jeżeli kolejka jest pusta (*underflow*)*:* nie ma nowych danych do odczytu, wtedy wątek może czekać na wstawienie danych (blokowanie) lub zrezygnować i przejść do wykonywania innych operacji.

Przykład zastosowania kolejki

```
// Wątek A – „producent”

while (1) {  // nieskończona pętla
   x = pobierz_dane();
   queue.put(x);
   // tutaj przechodzimy dopiero
   // gdy zapis się powiedzie
}
```

```
// Wątek B – „konsument”

while (1) {  // nieskończona pętla
   x = queue.get();
   // tutaj przechodzimy dopiero
   // gdy odczyt się powiedzie
   przetworz_dane(x);
}
```

Optymalizacja kolejek

- Programista powinien zadbać o zrównoważenie czasu wykonywania wątków.
- Kolejka nie powinna być nigdy ani pełna, ani pusta.
- Jeżeli kolejka będzie zapisywana szybciej niż odczytywana:
  - przepełnienie kolejki *(overflow),*
  - dane, których nie udało się zapisać w kolejce, zostają utracone.
- Jeżeli kolejka będzie odczytywana szybciej niż zapisywana:
  - efekt niedopełnienia kolejki (*underrun*), „zagłodzenie” wątku (*starvation*),
  - marnowanie cykli procesora – wątek nie ma danych, które mógłby przetwarzać.

Zdarzenia

- Czasami potrzebujemy wstrzymać wątek do czasu aż pewien zasób będzie dostępny.
- Zdarzenie (*event*) lub zmienna warunkowa (*condition variable*) jest sygnałem przesyłanym do innego wątku.
- Może mieć dwa stany: włączone lub wyłączone (*on/off*).
- Wątek może „zasnąć” i czekać na wystąpienie zdarzenia.
- Inny wątek może wysłać sygnał o dostępności zasobów, ustawiając zdarzenie na włączone.
- Czekający wątek zostaje wybudzony gdy otrzyma zdarzenie.
- Po odebraniu zdarzenia, wątek powinien je wyłączyć.
- Zdarzenie eliminuje konieczność aktywnego sprawdzania czy są nowe dane.

Przykład zastosowania zdarzeń

```
// Wątek A – wyższy priorytet

mutex.lock();

int i;
for (i = 0; i < 512; i++) {
    tablica[i] = pobierz_dane();
}

mutex.unlock();

event.enable(); // włączenie zdarzenia
```

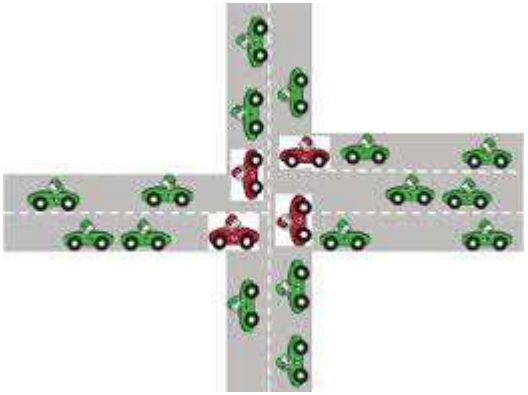
```
// Wątek B – niższy priorytet

event.wait(); // czekamy na zdarzenie

mutex.lock();

int i;
for (i = 0; i < 512; i++) {
    przetworz_dane(tablica[i]);
}

mutex.unlock();

event.disable();  // wyłączenie
```

Przerwania

- Przerwanie sprzętowe (*hardware interrupt*) jest sygnałem generowanym przez sprzęt, np. gdy pojawią się nowe dane na interfejsie wejściowym. Nie angażuje procesora.
- Programista pisze procedurę obsługi przerwania, realizowaną zwykle jako wątek o wysokim priorytecie. Procedura ta powinna działać możliwie krótko, aby nie blokować pozostałych wątków.
- Przerwania mają wyższy priorytet niż zwykłe wątki, dlatego obsługa przerwania wywłaszcza inne wątki.
- Przerwania niemaskowalne – nie mogą zostać wyłączone w programie, np. sygnał RESET. Przerwania maskowalne mogą być wyłączone.
- Przerwania programowe (*software interrupt*) – generowane przez programistę, na podobnej zasadzie jak sprzętowe (ale z udziałem procesora).

Zakleszczenie wątków

- Pisząc programy wielowątkowe musimy bezwzględnie unikać takich sytuacji:
  - wątek A trzyma muteks 1 i próbuje zablokować muteks 2 aby zapisać dane,
  - wątek B trzyma muteks 2 i próbuje zablokować muteks 1 aby dokończyć przetwarzanie danych.
- Powstaje zakleszczenie wątków (*deadlock*): żaden z wątków nie może kontynuować działania i program się zawiesza.
- Taka sytuacja jest winą programisty, ale jest trudna do zdiagnozowania ponieważ nie musi wystąpić za każdym razem.



Programy niedeterministyczne

- Program jednowątkowy jest deterministyczny. Kolejność wykonywania instrukcji w programie jest przewidywalna, taka sama przy każdym uruchomieniu programu.
- Programy wielowątkowe są niedetrministyczne:
  - nie da się przewidzieć w którym momencie nastąpi przełączenie wątku,
  - zależności czasowe między wątkami mogą być inne przy każdym kolejnym uruchomieniu programu,
  - występują wyścigi (*race*) – wątki rywalizują o dostęp do zasobów, wykonywanie programu może zależeć od tego czy wątek zdąży przejąć zasób przed innym,
  - z tego powodu program wielowątkowy zawierający błąd może np. zawiesić się jeden raz na dwadzieścia uruchomień.
- Debugowanie programów wielowątkowych jest bardzo trudne.
- Programista musi minimalizować ryzyko wystąpienia konfliktowych sytuacji.

Systemy operacyjne na DSP

Przykład – procesory Texas Instruments.

- System operacyjny czasu rzeczywistego, pod różnymi nazwami: DSP/BIOS, SYS/BIOS, TI-RTOS.
- Umożliwia uruchamianie wielowątkowych programów na jednym rdzeniu DSP.
- Zapewnia mechanizmy synchronizacji wątków i wymiany danych między nimi.
- System jest dostarczany w ramach pakietu *Processor SDK*.
- Kod systemu operacyjnego jest łączony z kodem programisty w jeden program wykonywalny.

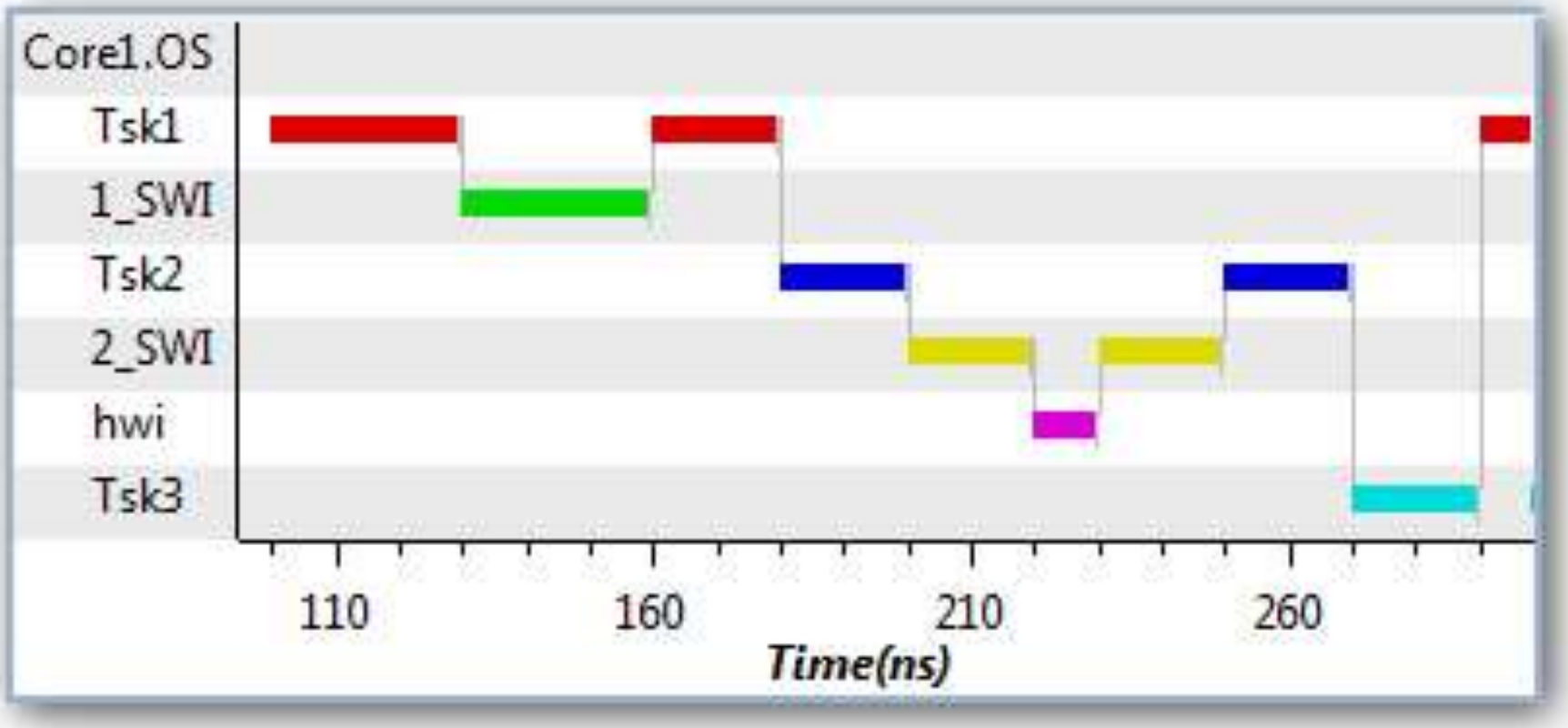
Przykład systemu: TI SYS/BIOS

Terminy stosowane przez DSP/BIOS i SYS/BIOS:

- *interrupt* (przerwanie) – wątek obsługi przerwania sprzętowego i programowego,
- *task* (zadanie) – wątek nie związany z przerwaniami,
- *idle loop* (pętla bezczynności) – wątek tła o najniższym priorytecie, uruchamiany tylko wtedy gdy wszystkie inne wątki są wstrzymane,
- *gate* (bramka) – muteks, blokuje dostęp do zasobów,
- *semaphore* (semafor) – licznik dostępnych zasobów,
- *mailbox* (skrzynka pocztowa) – struktura do wymiany komunikatów między wątkami,
- *queue* (kolejka) – struktura do przekazywania danych między wątkami,
- *memory section manager* – moduł zarządzania dynamicznie alokowaną pamięcią,
- *pipe* (potok), *stream* (strumień) – wymiana danych między interfejsami a pamięcią.

Przykład zarządzania wątkami przez system operacyjny

W kolejności priorytetów: wątki obsługi przerwań sprzętowych (*hwi*) i programowych (*1_SWI*, *2_SWI*) oraz wątki utworzone przez programistę (*Tsk1*, *Tsk2*, *Tsk3*)

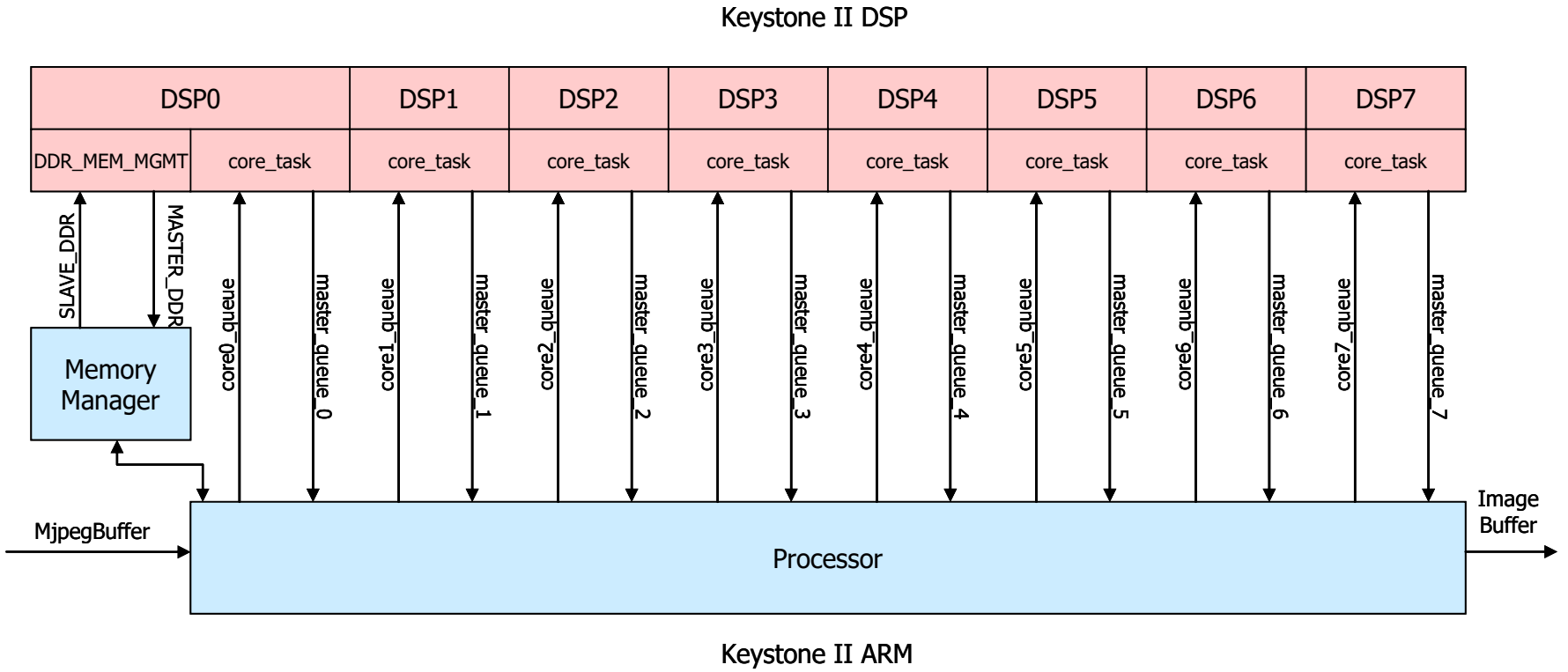


Systemy na procesorach hybrydowych

- Hybrydowe procesory sygnałowe – zawierają rdzenie ARM i DSP.
- Przykład: TI Keystone C66x+ARM (1 × ARM, 8 × DSP).
- Przetwarzanie na takim procesorze odbywa się w trybie *master-slave* (zarządca i wykonawca).
- Na procesorze ARM jest uruchamiany system operacyjny:
  - dedykowany dla procesora (np. TI-RTOS),
  - lub typowe systemy (np. Linux) z jądrem czasu rzeczywistego (np. Yocto).
- Program na rdzeniu ARM: zarządca, kieruje dane do wykonawców, odbiera wyniki.
- Program na rdzeniach DSP: wykonawca, przetwarzanie danych.
- Do wymiany danych między ARM a DSP służą kolejki.
- Kilka rdzeni DSP umożliwia zrównoleglenie obliczeń (SIMD).

Systemy na procesorach hybrydowych

Wymiana danych między rdzeniami ARM i DSP za pomocą kolejek:



Podsumowanie

- Proste programy na procesorach sygnałowych, takie jak na laboratorium ZPS - programowanie *bare metal* (jednowątkowe) wystarcza.
- Jeżeli program spędza zbyt dużo czasu bezczynnie czekając na dane i brakuje cykli do obliczeń – potrzebny jest system operacyjny.
- Programowanie wielowątkowe jest znacznie trudniejsze niż jednowątkowe – wiele rzeczy może pójść źle, a znalezienie przyczyny problemu jest często bardzo trudne i czasochłonne.
- Należy dobrze przemyśleć podział zadań pomiędzy wątki.
- System operacyjny wykonuje za nas operacje zarządzania wątkami i zasobami.
- Procesory hybrydowe DSP+ARM dają potencjalnie większą wydajność i elastyczność, kosztem zwiększonego narzutu przetwarzania.

Podsumowanie

Bardzo częsta odpowiedź studentów na kolokwium:

> *„W programie wielowątkowym wiele wątków działa jednocześnie, dzięki czemu program wykonuje się szybciej.”*

Jest to nadmierne uproszczenie.

- *„Wiele wątków działa jednocześnie”*: są jednocześnie uruchomione, ale na jednym rdzeniu procesora działa (wykonuje instrukcje) tylko jeden wątek, pozostałe „śpią”.
- *„Program wykonuje się szybciej”* – niekoniecznie. Jeżeli program nie musi czekać na dane, to wielowątkowy program może być wolniejszy (ze względu na narzut).
- Ale jeżeli program często czeka na dane (blokowanie programu przez I/O), możemy wykorzystać przełączanie wątków aby zoptymalizować wykorzystanie zasobów procesora. Wtedy rzeczywiście program wielowątkowy wykonuje się szybciej niż jednowątkowy, bo redukujemy czas bezczynności procesora.