OPERATING SYSTEMS OF DIGITAL SIGNAL PROCESSORS

Author: Grzegorz Szwoch

Gdańsk University of Technology, Department of Multimedia Systems

Programming without OS

- Simple DSP programs do not need an operating system.
- Such a program runs programmed operations sequentially.
- This method of programming is called *bare metal*
  - directly on a processor.
- There is a drawback: if a program waits for data, it cannot perform any operations (this is I/O blocking).
- For more complex programs, this is not optimal: available processing time is turned into idle time and wasted.

Input/output operations

- Input/output (I/O) operations: getting data from inputs, sending results to the output.
- Waiting for data causes the program to stop.
- It is problematic when the program gets the data from multiple inputs.
- We want to do computation while waiting for new data.
- The solution: dividing the program into threads.
- An operating system is needed to manage (“govern”) threads.
- On DSPs, the operating system is usually provided by the manufacturer, OS code is merged with the program.

Program and threads

- Program – the complete code of operations performed on a DSP. One DSP (or one DSP core) = one program.
- Thread – a separated section of the program.
- A program can have multiple threads.
- This is called multithreading or concurrency.
- Threads compete for resources: processor cycles, memory.
- Operating system (OS) on a DSP: the main thread that manages other threads and their access to the resources.

Example threads

Thread 1 – reads data:

- reads data, e.g. from a sensor,
- passes data to thread 2,
- sleeps, waiting for new data.

Thread 2 – processes data:

- gets data from thread 1,
- processes data and sends the result to the output,
- sleeps, waiting for new data.

Concurrency and parallelism

These terms are often mixed.

Concurrency:

- there are multiple threads,
- the number of active threads that are running is ≤ number of available processors/cores,
- a DSP with a single core: one thread is running, the rest is sleeping.

Parallelism:

- multiple threads are running at the same time,
- requires multiple DSPs or a multicore DSP.

Types of multitasking

Cooperative multitasking:

- a thread must stop so that another thread can start,
- used in programs in which the programmer has full control on all threads,
- sometimes used in embedded systems.

Preemptive multitasking:

- an operating system manages the threads,
- any thread may be stopped (preempted) by the OS at any time, so that another thread can run,
- more efficient in terms of resource usage,
- used on PCs and in most DSP programs.

Context switch

- Context – a running thread and resources it uses.
- Context switch – the running thread is stopped and the resources are assigned to another thread.
- Priority – a number that determines hierarchy of threads.
- A context switch usually occurs when:
  - a thread goes to sleep, waiting for data, and yielding the resources voluntarily,
  - a higher priority thread demands resources, so the currently running, lower priority thread is stopped (preempted).

Memory conflicts

Threads compete for resources which may cause conflicts.
An example of a conflict situation:

Thread 1 (higher priority):

- (sleeps)
- wakes up
- writes new data to memory
- goes to sleep
- (sleeps)

Thread 2 (lower priority):

- reads the first part of a buffer in memory
- preempted by Thread 1
- (waits)
- resumes execution
- reads the second part of a buffer in memory - the content is garbled!

Mutex

- Mutex is an object that allows for an exclusive access to memory by a thread.
- A thread that wants access to shared resources must obtain the mutex and lock it.
- If a mutex is locked by another thread, the first thread must wait, or it can perform other operations while waiting.
- Mutex must be unlocked after the operations are completed.
- Critical section – a section of code protected by a mutex, which must be executed completely by a single thread.
- Mutexes slow down program execution, so they should be used only if needed.

Mutex - an example

Thread 1:

- locks the mutex
- reads the buffer
- unlocks the mutex

Thread 2:

- …
- waits for the mutex
- locks the mutex
- writes new data to buffer
- unlocks the mutex

Semaphore

- Semaphore is an object that allows access to limited resources from multiple threads.
- A semaphore has a counter.
- Locking a semaphore decreases the counter by 1, unlocking increases the counter by 1.
- Access to the protected resource is not possible when the counter is zero.
- An example: there are 5 buffers in memory
  - current counter: 1 (four buffers are already taken)
  - thread #1 takes the last buffer, the counter is 0
  - thread #2 cannot take the buffer, it must wait until another thread releases the buffer and the counter is 1.

Queue

- Threads must pass data between them.
- Queue is a structure that allows putting and getting data.
- A typical scheme is “producer-consumer”:
  - one thread (the producer) generates data (e.g. gets them from the input) and puts them in the queue,
  - another thread (the consumer) gets data from the queue and processes them.
- A queue is usually protected with an internal mutex.
- Data cannot be put into the queue if it is already full.

Queue example

Thread 1 (producer):

- reads input data
- puts data into queue
- waits for new data

Thread 2 (consumer):

- gets data from queue
- waits if the queue is empty
- processes the data

Optymalizacja kolejek

- A programmer should balance execution time of threads.
- A queue should never be full or empty.
- If a queue is written to more often that it is read:
  overflow occurs, data may be lost.
- If a queue is read more often than it is written:
  underrun (starvation) occurs, processor cycles are wasted.

Events

- Events are notifications sent to other threads.
- Usually, they notify a thread that new data are available.
- Event may be “on” or “off”.
- A thread may go to sleep and it may be woken up by an event that is “on”.
- When a thread receives the event, it should turn off that event.
- Events eliminate the need for polling – active checking for data availability.

Interrupts

- Hardware interrupt is a notification generated by hardware, e.g. when an interface receives new data.
- A thread that is assigned to a given interrupt, handles that interrupt (it is called automatically).
- Software interrupts may be generated by a programmer.
- Interrupts have higher priority than threads, so interrupt handling stops other threads.
- Non-maskable interrupts (NMI) cannot be turned off
(for example: RESET signal).

Deadlock

This case must not happen:

Thread 1:

- locks mutex_A
- waits for mutex_B

Thread 2:

- locks mutex_B
- waits for mutex_A

- Both threads wait for resources that cannot be obtained, because they remain locked.
- This is an example of a deadlock.
- Usually, a deadlock hangs the program.

Concurrent programming problems

- It is said that concurrent programs are non-deterministic, because the results of their execution depend on the order of instructions from different threads and the time of context switching.
- Race – a successful acquisition of resources depends on whether a thread is faster than another thread.
- The order of executed instructions may be different each time the program is run.
- Deadlock may occur e.g. every 20th run of the program.
- Debugging concurrent programs is very hard.
- The programmer must reduce the risk of all conflicts.

Operating systems on DSP

Example – Texas Instruments processors

- Real-time operating system, under various names: DSP/BIOS, SYS/BIOS, TI-RTOS.
- Allows for execution of multithread programs on a single-core DSP.
- Provides means of thread synchronization.
- The OS is merged with the signal processing code into a single executable program.

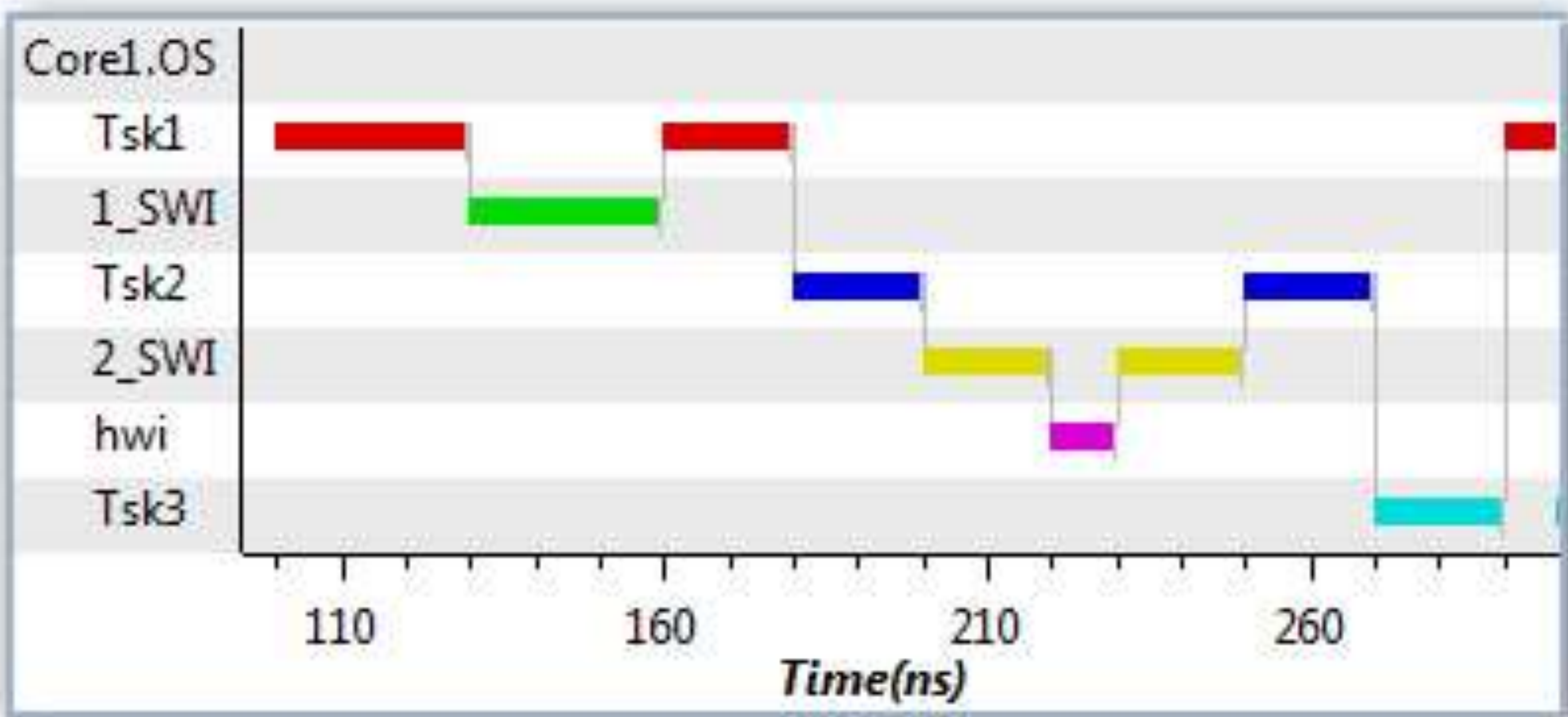
Terms

DSP/BIOS and SYS/BIOS use the following terms.

- *Interrupt* – a thread that handles interrupts.
- *Task* – a thread that processes data.
- *Idle loop* – a lowest priority thread that runs I background only if no other thread is active.
- *Semaphore* – semaphore, *Gate* – mutex.
- *Mailbox* – a structure for exchanging messages between threads.
- *Queue* – a structure for data exchange between threads.
- *Memory section manager* – a module that manages dynamically allocated memory.
- *Pipe, stream* – data exchange between interfaces and memory, through DMA.

Example of thread management

Threads for handling hardware (hwi) and software (SWI) interrupts and data processing threads (Tsk).

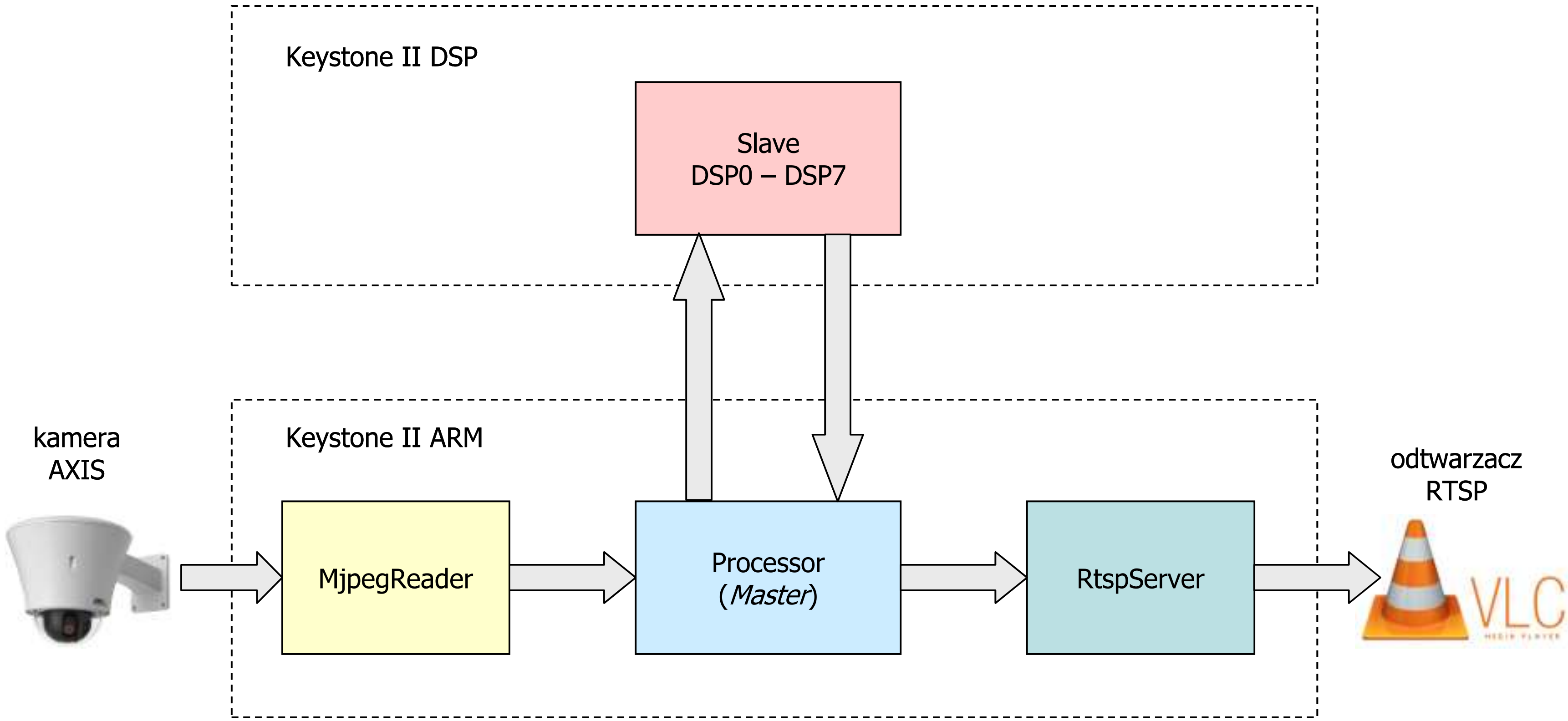


OS on hybrid processors

- Hybrid DSPs – composed from ARM and DSP cores.
- Example: TI Keystone C66x+ARM.
- Processing on hybrid OS uses the master-slave paradigm.
- The ARM cores run the operating system:
  - special OS for the processor (e.g. TI-RTOS),
  - or Linux with real-time kernel (e.g. Yocto).
- Program on ARM cores: master, passes data to slaves and collects the results.
- Program on DSP cores: slave, performs the processing.
- Queues are used to exchange data between master and slaves.

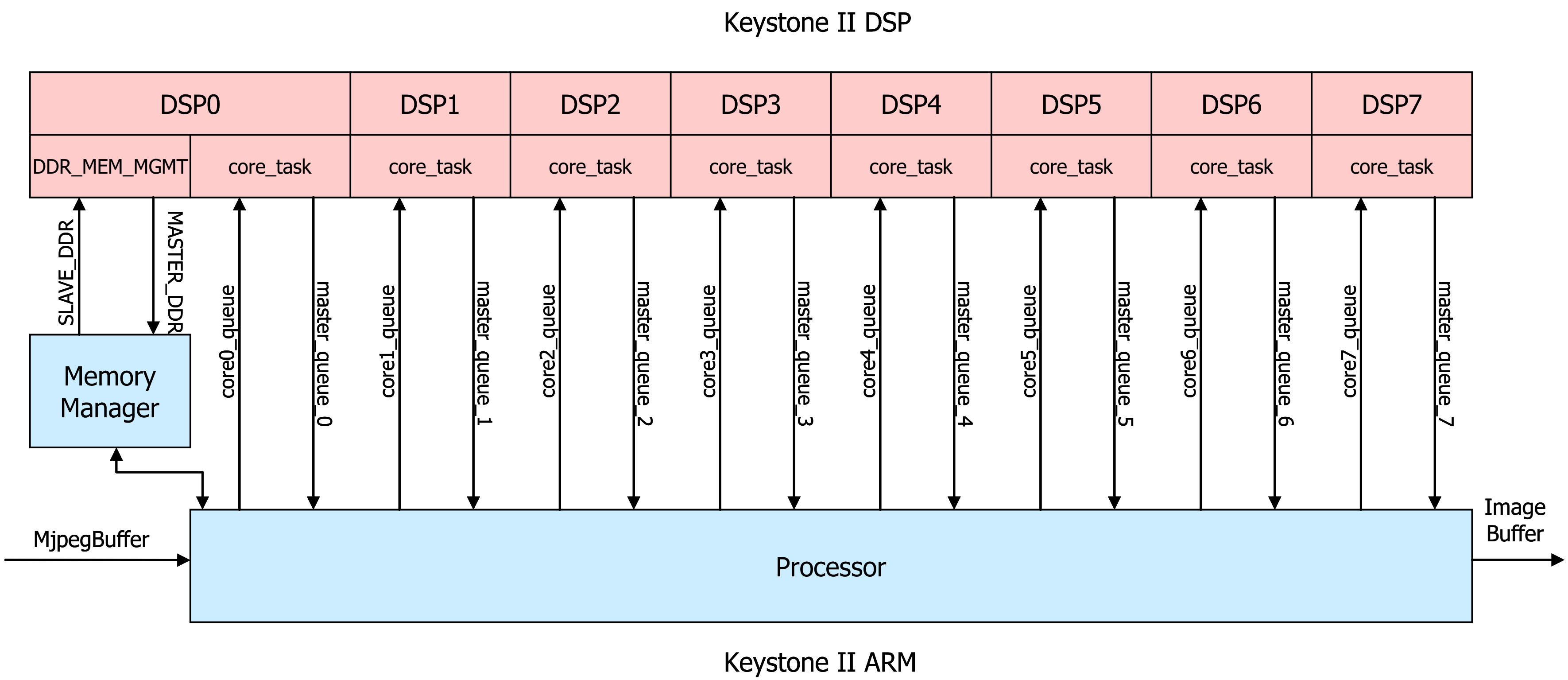
OS on hybrid processors

Example: processing of a camera video. Each DSP core processes a piece of the image.



OS on hybrid processors

Data exchange with queues



Summary

- Simple DSP programs, such as the one written in the course project, do not need an OS, bare metal is enough.
- If a program spends too much time waiting for data and there is not enough cycles for processing – an OS is needed.
- Concurrent programming is much more difficult than writing standard programs, many things can go wrong.
- Dividing the processing into threads must be carefully planned.
- The operating system manages threads for a programmer.
- Hybrid DSP+ARM processors have potentially better efficiency and flexibility, but the processing overhead is higher, and the programming is more difficult.