Userspace physical memory buffers kernel driver design

Mobile S/W Platform Lab.

2009.04.21

Executive summary

- **Problem – current memory management in multimedia devices is:**
  - difficult to maintain, does not integrate easily with open source solutions
  - wastes a lot of memory
  - **INSECURE** – can be very easily used for exploits or root kit purposes

- **Proposed solution:**
  - shared memory buffers based on user space addresses and translation layer
  - physical memory manager

- **Advantages:**
  - **transparent and easy usage** both in userspace and kernel drivers
  - memory is allocated only on demand
  - **high security** based on existing kernel infrastructure
  - **transparent integration with open source** solutions (no need to patch existing drivers)
  - implementation of multimedia drivers and OpenGL can be **faster and cleaner**
  - **ready to use** – implemented and tested with 2D graphics accelerator driver

Revision History

Ver.	Date	Description	Author	Reviewer
0.1	2009/04/21	Initial Version	Marek Szyprowski, Pawel Osciak	

Contents

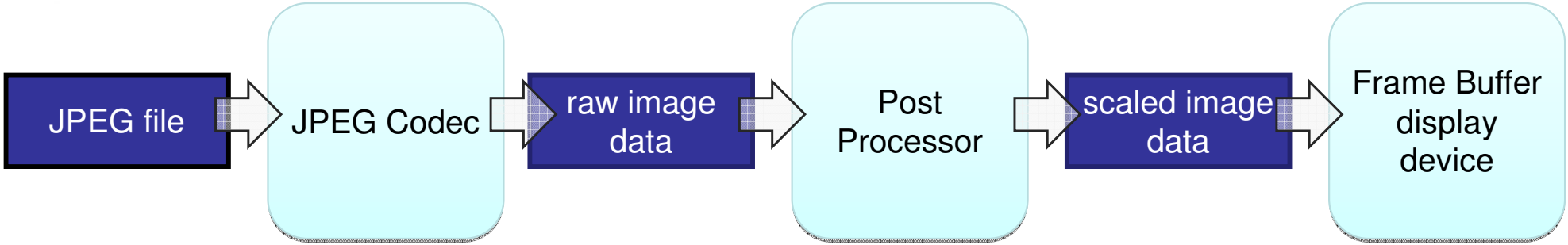
S3C64XX SOC – Hardware capabilities

- **Various multimedia devices are available:**
  - Frame Buffer, 3D graphics accelerator
  - 2D graphics blitter, Post Processor (color space converter/scaler)
  - JPEG codec, Multi-Function Codec
  - Camera interface

- **Common features of multimedia devices:**
  - all require one or more memory buffers for input or/and output data
  - the buffers must be in system memory, devices can access it directly
  - no scatter-gather functionality, so memory buffers must be contiguous in physical memory

Typical use case of multimedia devices

- **Usually multimedia devices are used in pipeline style:**
  - output buffers from one device are often used as input buffers for other devices
  - theoretical example:



Legend:

memory data buffer

multimedia device

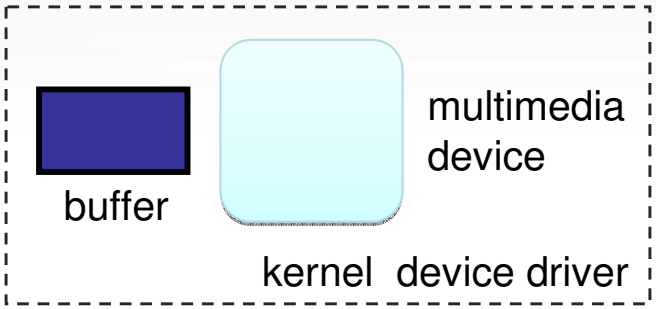
data flow

Pipeline data processing and kernel/user space

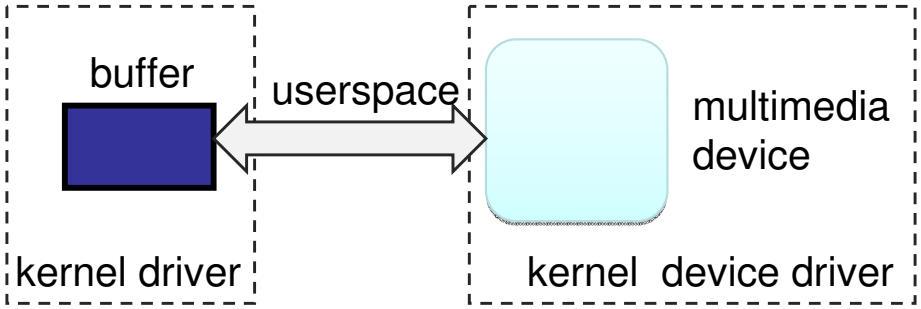
- Processing of data through a pipeline must be properly separated between kernel drivers and user applications
- Each part of a pipeline must either be a kernel driver or a userspace application module
- Connections between different devices are set up by userspace applications
- The separation should be done in a way that will have a minimum impact on the throughput of all the possible use cases of different multimedia devices
- Additional problem – **drivers for some of the devices are already written** by open source community and might not match the solutions used in other multimedia devices

Pipeline data processing and kernel/user space

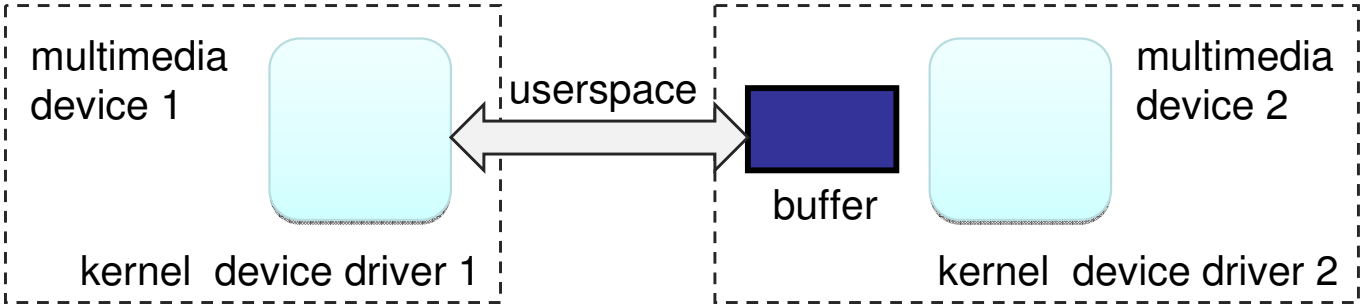
- **Types of relations between drivers and memory buffers**
  - as a part of kernel driver:



  - as a separate driver:

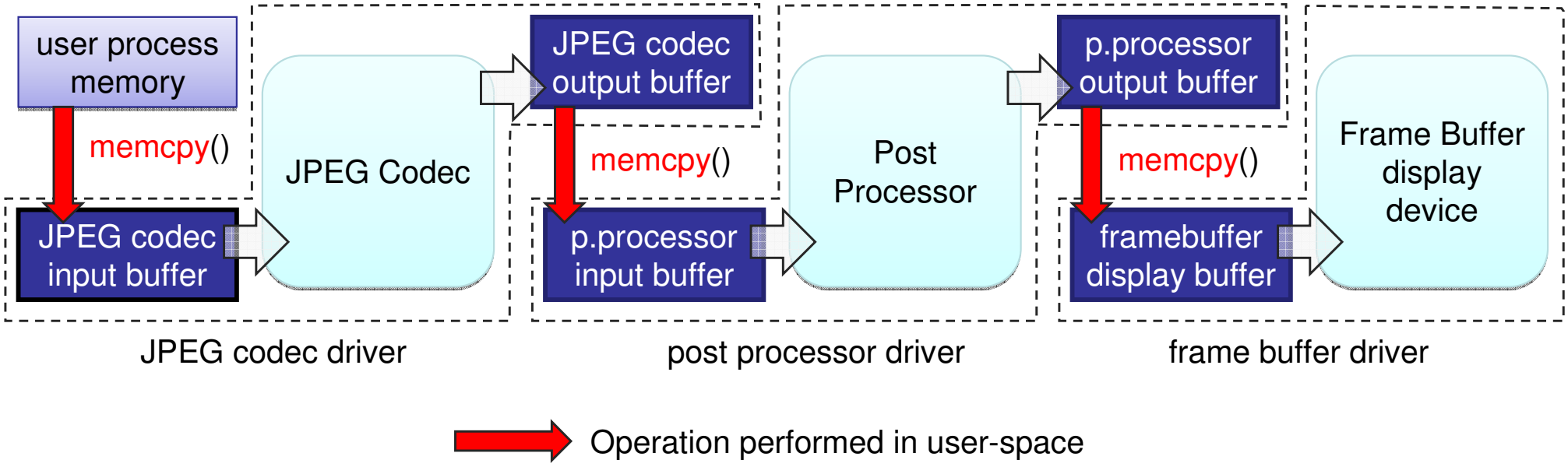


  - mixed:



Typical Linux kernel driver design

- **Kernel drivers work in kernel address space and have unlimited access to all system and hardware resources**
- **Typically each driver allocates its own input and output buffers and gives access to them to user-space (usually by special mmap() device call)**
- **Copying of buffers is done in user-space**
  - device drivers can only are opened by user-space applications
  - only user-space applications have access to both input and output buffers
- **In this architecture the pipeline example will look like this:**



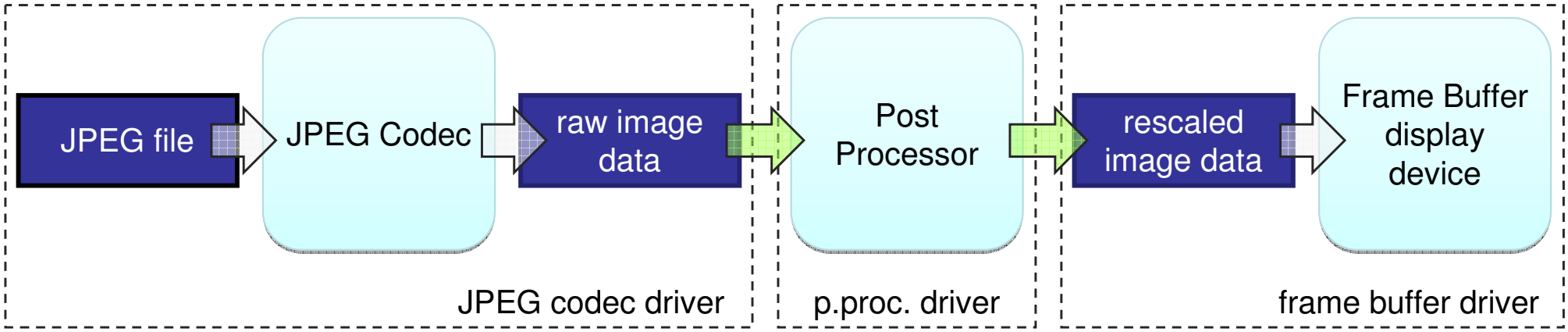
Typical Linux kernel driver design

- **Advantages of standard kernel drivers:**
  - **SECURITY**
    - only device drivers allocate buffers and set up hardware blocks to access them – driver has all the required information and can check whether a buffer has the right properties
    - innocent or foreign memory cannot be accessed in any case (besides driver bugs)
  - **simple user-space driver usage**
    - user-space applications usually open device special file, then call mmap() function and have all required buffers accessible from their address space

- **Disadvantages:**
  - **REDUCED SPEED**
    - for each input/output buffer pair a memory coping operation needs to be done
    - this lowers total pipeline throughput and heavily increases CPU usage (memory copying is usually done by CPU)
  - **memory requirements**
    - 2 times more memory is required as almost all buffers are duplicated
    - allocating physically contiguous memory is a real problem and should be reduced to absolute minimum (physical memory fragmentation issue)

Better solution - shared data buffers

- Idea is simple – give ability to share buffers between different drivers
- No data copying is needed, high speed when processing data in pipeline
- Device drivers need to be aware of buffer sharing – it should be possible to set a driver to use an external buffer
- Example:



- Different implementations possible :
  1. based on buffer physical address
  2. based on buffer ID
  3. based on a user-space address to buffer – **our proposed solution**

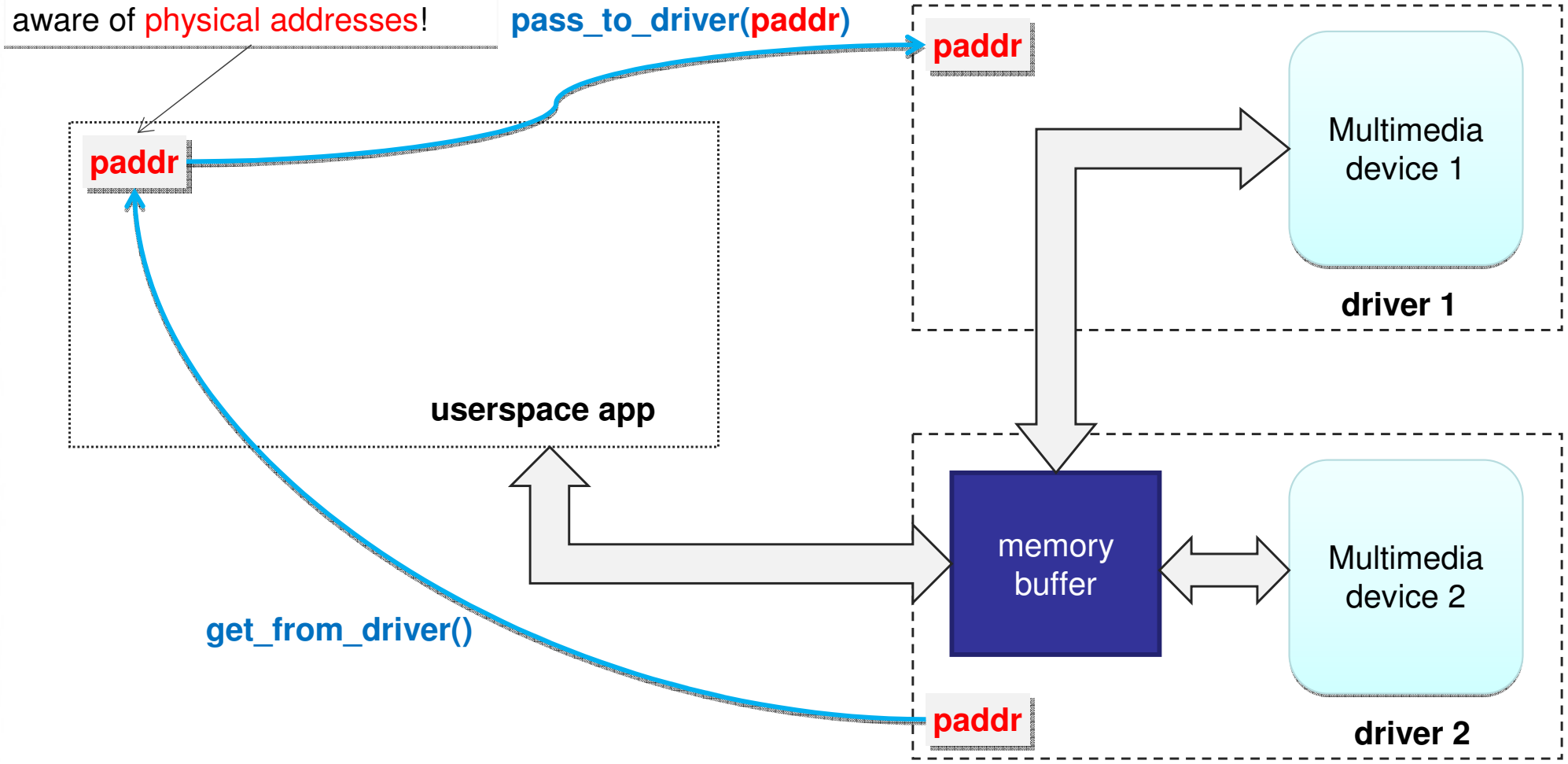
1. Implementation based on physical addresses

- **Simplest approach**
- **Drivers can allocate or reserve memory for all its buffers on boot, but this will result in a huge memory waste as not all of it be used simultaneously.**
- **Only device drivers that are planned to be used in processing pipeline need to be modified:**
  - get_phy_buffer() and set_phy_buffer() calls need to be added
  - a possibility of using external buffers might be required
- **User-space application must decide which buffers are used in processing pipeline – the buffer allocated by the driver itself or the one acquired from another driver**

1. Implementation based on physical addresses

This paddr can be **anything**!
If it is wrong or deliberately changed to cause damage, the driver will **overwrite kernel memory, other applications, or anything in the system** !

Userspace application has to be aware of physical addresses!



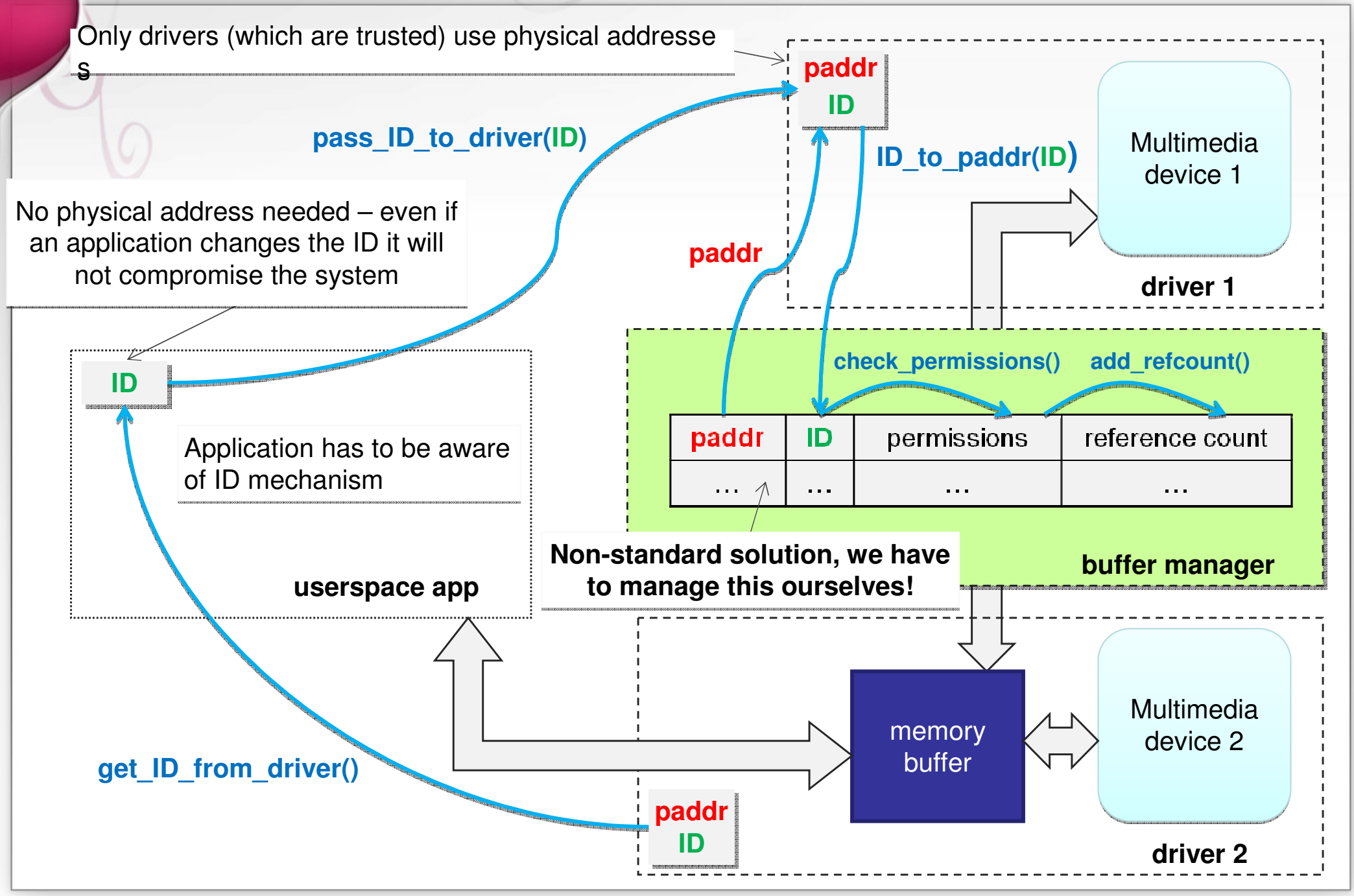
1. Implementation based on physical addresses

- **Disadvantages:**
  - **MAJOR security issues**
    - user-space application can provide ANY physical address to any device
    - with specially prepared data it is possible to modify and compromise the kernel or any other user-space application memory
    - this solution is **completely inacceptable** in DRM-aware context where **only signed and trusted code may be executed**
    - no real buffer management – it is not possible to lock a buffer based on its physical address and thus prevent a user from releasing it
  - There is **no way to easily mix with open source drivers** (which are not aware of such buffers) with proprietary ones

2. Implementation based on buffer ID

- **This solution solves problems found in the previous solution**
- **Instead of physical memory address, a special *ID* or '*key*' is used**
- **Buffer manager is required:**
  - each driver registers all its buffers in buffer manager and gets unique ID for them
  - driver exports only buffer ID to userspace (not a physical address)
  - buffer manager keeps track on use count of each buffer and refuses to unregister buffer if it is still in use
  - buffer manager checks if user that requested access to a buffer has in fact rights to access it
- **It is not possible to access innocent or foreign memory.**
- **Disadvantages:**
  - All device drivers used in pipeline must be aware of this solution.
  - There is no possibility to easily mix opensource drivers (that are not aware of shared buffers) with proprietary ones. Opensource drivers need to be patched to support buffer sharing.
  - Open source community has not provided any similar solution till now, so open source drivers are not compatible with such solution.

2. Implementation based on buffer ID



3. Implementation based on a user-space address 1/6

- **Our proposed solution**

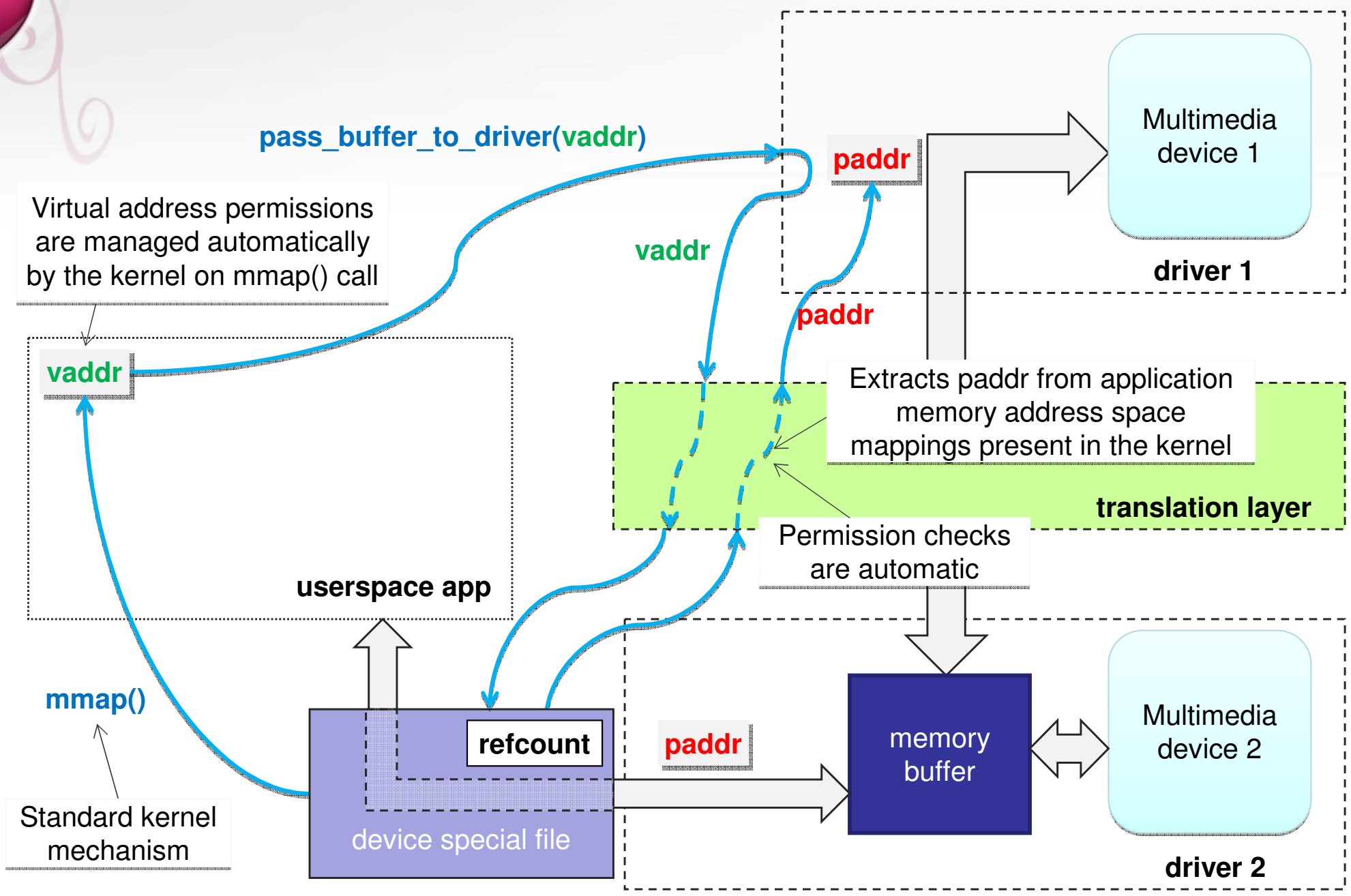
- **User-space application memory address space usually consist of the following 3 parts:**
  - real memory (stack and heap)
  - memory mapped disk files (binaries, shared libraries)
  - **memory mapped special devices – these regions are usually directly mapped to physical memory used as device driver buffers**

- **Device drivers can access physical memory** that corresponds to the specified process virtual memory area.

- This technique is called *zero-copy user space access.*

- Before physical memory can be directly accessed it must be **locked** properly, to ensure that kernel or other drivers will **not free or relocate** it.

3. Implementation based on a user-space address 2/6



3. Implementation based on a user-space address 3/6

- **To address these requirements a special transformation layer is required, that:**
  - translates userspace pointer and size pair into a physical memory address
  - uses process memory address space map to find all needed information
  - locking of specified userspace memory area:
    - in case of memory mapped special devices – it can directly increase the use count of special file and return the address of the physical memory
    - in case of real process memory – it can lock user pages to force them to stay on the same physical pages and return the address of that physical memory
  - if accessing physical memory is not directly possible, the translation layer can create a shadow buffer of that region and copy data to/from it before and after device operation
- **A physical memory allocator is also needed**
  - there must be a well defined way of allocating physical contiguous memory (at least for allocating shadow buffers)
  - there must be a special driver that will provide a method of allocating contiguous memory for userspace applications, that are aware of fact that memory allocated in such a way will work faster with multimedia devices (no shadow buffer is needed)

3. Implementation based on a user-space address 4/6

Advantages:

- **Easy usage from user-space**
  - user does not need to know how the memory area has been allocated
  - usage scenarios:
    - user can mmap() frame buffer or any other multimedia device and use the acquired pointer to it directly as a target for the other multimedia device
    - however as a compatibility fallback, user can allocate any memory buffers (using malloc() or other allocator function) for internal use and specify the acquired virtual addresses as source or target for all multimedia devices
    - for best performance for internal application use, user should also use buffers allocated by contiguous memory allocator to ensure that no shadow buffers will be required

3. Implementation based on a user-space address 5/6

Advantages (continued):

- **Easy usage from kernels drivers**
  - well-defined layer that provides a method of getting physical contiguous memory buffers from user-space
  - **multimedia device drivers do not have to care** how the buffers passed by user-space applications have been allocated - **the translation layer ensures** they will get a physical memory area with the data from the user-space
  - the API is **very simple** and consist of 3 functions:
    - prepare_buffer() – sets up buffer for transactions (locks user space memory)
    - sync_buffer() – synchronizes buffer with CPU and user-space (makes buffer valid before device DMA read or after device DMA write)
    - release_buffer() – reduces buffer use count and frees all allocated resources

- **No memory is wasted – buffers are allocated on demand and have the size that is equal to the size of processed data.**

3. Implementation based on a user-space address 6/6

Advantages (continued):

- **SAFE and SECURE:**
  - if a userspace application manages to map specified memory to its address space, it means all access right are already checked.
  - it is not possible to access any unallocated memory, because any invalid pointer can be easily detected.
- This solution **transparently integrates with open source drivers** that provide internal buffers by mmap() call. The **drivers do not need to be aware of shared buffers** if their buffers are only used either as source or destination buffer.

  This way there is no need to make any changes to open source drivers, so maintenance is much easier.

Physical memory manager

- **Provides a common way to allocate physical, continous memory to either kernel drivers or user space applications.**

- **User-space application can allocate continous physical memory by calling mmap() on the special device.**
- **Memory allocated this way will be detected by translation layer and can be directly used as a buffer by any multimedia device.**

- **Memory manager can create a *SYSV SHMEM* area for allocated memory buffer.**
- ***SYSV SHMEM* areas can be directly used in *X*-environment (*XShm* extension)**
  - this makes it possible to have **transparent, zero-copy blitting directly from userspace** application through all X-protocol layers and XServer to FrameBuffer device!
  - especially **applicable for integrating OpenGL with X-Server** and userspace applications

Ready to use

- **Proposed solution:**
  - has already been implemented
  - has already been tested and works properly
  - used in 2D graphics accelerator driver and allowed to integrate it with open source frame buffer driver without the need of patching it.

Summary

- **A uniform memory management for multimedia devices is needed**
- **The existing software does not provide required features**
- **The proposed solution – shared buffers based on userspace pointer address to buffer adresses most of the problems**
  - it is robust, fast and secure
  - can be transparently integrated with existing opensource drivers
- **There are drawbacks though**
  - adopting or rewriting some of the existing code (mainly multimedia device drivers) is required.

Thank You