Reusable Generic Look Ahead Multi threaded Cache

a case study for a high resolution player

Guillaume Chatelet - MikrosImage

Menu

- Meeting Duke
- A few playback issues
- A naive implementation and how to improve
- A briefer history of data sharing
- The current Cache design
- Demo
- Conclusion + Q&A

Duke

High performance, free, multi-platform player for the VFX industry

https://github.com/mikrosimage/duke

Context : player features

- fluid playback, quick seeking
- many formats - plugin based
  - file sequences : OpenEXR, DPX, Jpeg2k...
  - movies : h264, mpeg2, ... (in progress)
- stereoscopic & multi tracks (in progress)
- sound sync
- colorimetric & grading capabilities
- cmd line, server mode, integrated

Mission : perfect playback

What are we after ?

- realtime playback (direct to disk or RAM)
- perfectly fluid

How hard is it ?

- big images : 2K, 4K
- high dynamic : 10 bits log, 16 bits float
- many streams : stereoscopic, multi-tracks
- fast : 24, 25, 30 FPS

Mission : perfect playback

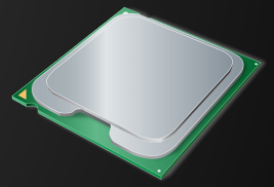
Image weight & throughput needed

layout	bits / chan	definition	image weight	25 FPS throughput
RGBA	8	2048x1152	9.44MB	236 MB/s
RGB	10	2048x1152	9.44MB	236 MB/s
RGBA	16 half float	2048x1152	18.9MB	472 MB/s
RGB	10	4096x2304	37.7MB	944 MB/s
RGBA	16 half float	4096x2304	75.5MB	1 887 MB/s

The classical trade-offs

IO versus CPU versus Quality

- Pressure over I/O
  - uncompressed images => (very) fast disks
    - DPX files
- Pressure over decoding power
  - compressed images
    - space lossless : png
    - space : sequences of jpeg, jp2k, openEXR...
    - space + time : mpeg2, h264, ProRes



Context : *classical trade-offs*

Time versus Space versus Quality

Space	**Time**	**Container**	**Format**
uncompressed	uncompressed	sequence	dpx, tga, pnm...
compressed lossless	uncompressed	sequence	*jpeg2K*, OpenEXR, png...
compressed lossy	uncompressed	sequence	*jpeg*, OpenEXR B44, ...
compressed lossy	compressed lossy	movie	Mpeg2, h264, ProRes...

So what kind of machines are we aiming at ?

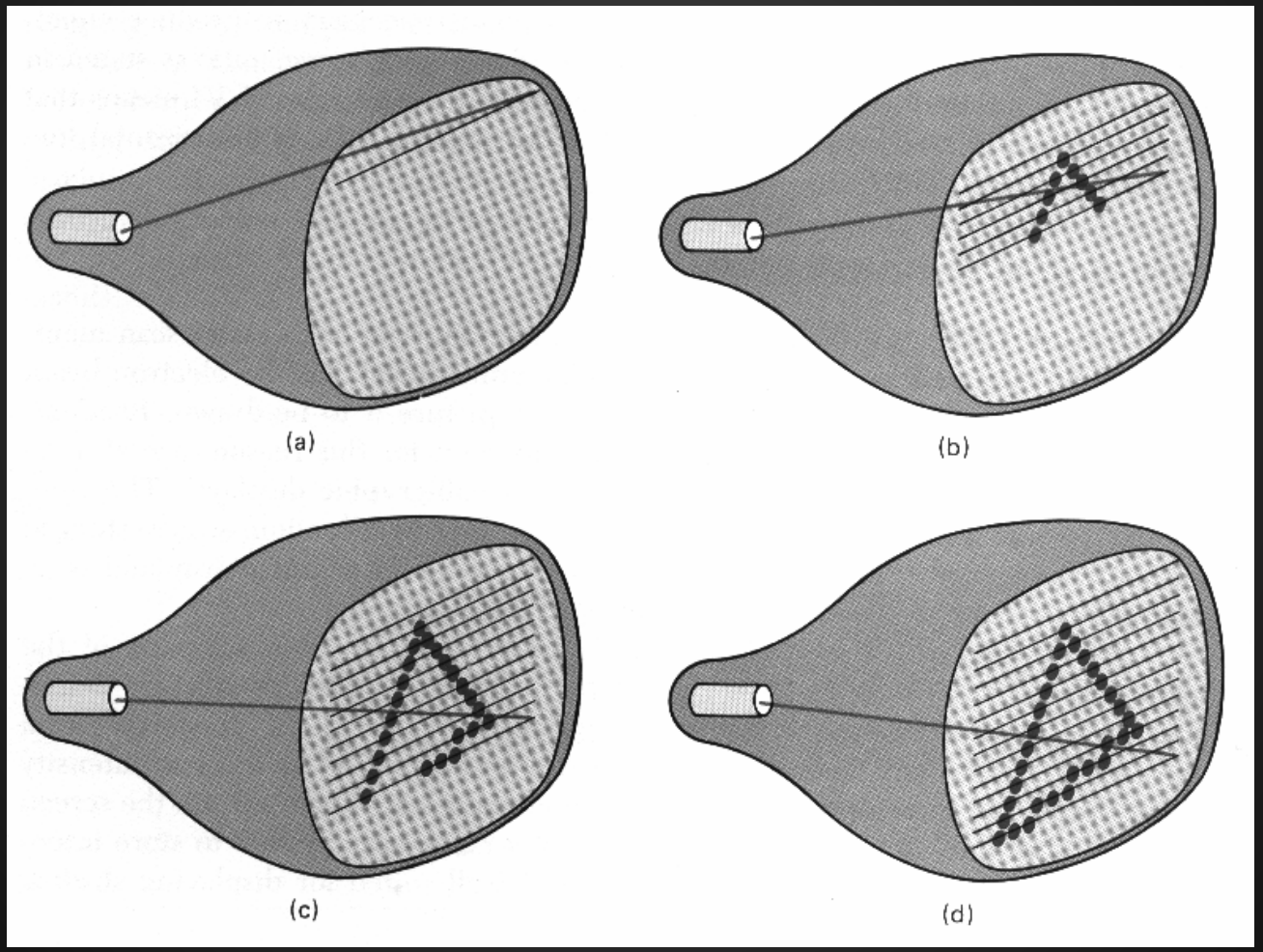
- 16-cores machine with dedicated storage and plenty of RAM ?
- Simple desktop machine with slow disks or reading over network?

I want it all ! I want it now !

Context : playback issues

- Image synchronization
  - Screen tearing
  - Jerky playback : framerate VS refreshrate
- Are we fast enough ?
  - Depends on the machine
  - IO throughput is limited
  - Latencies lurking everywhere
  - No realtime OS so we have to be foxy

Screen Tearing



Screen Tearing

2 buffer swap within a single raster scan

Screen Tearing

Jerky playback

Camera panning : slow regular movement
enforce same duration for each single frame

refreshrate must be a multiple of framerate

Jerky playback

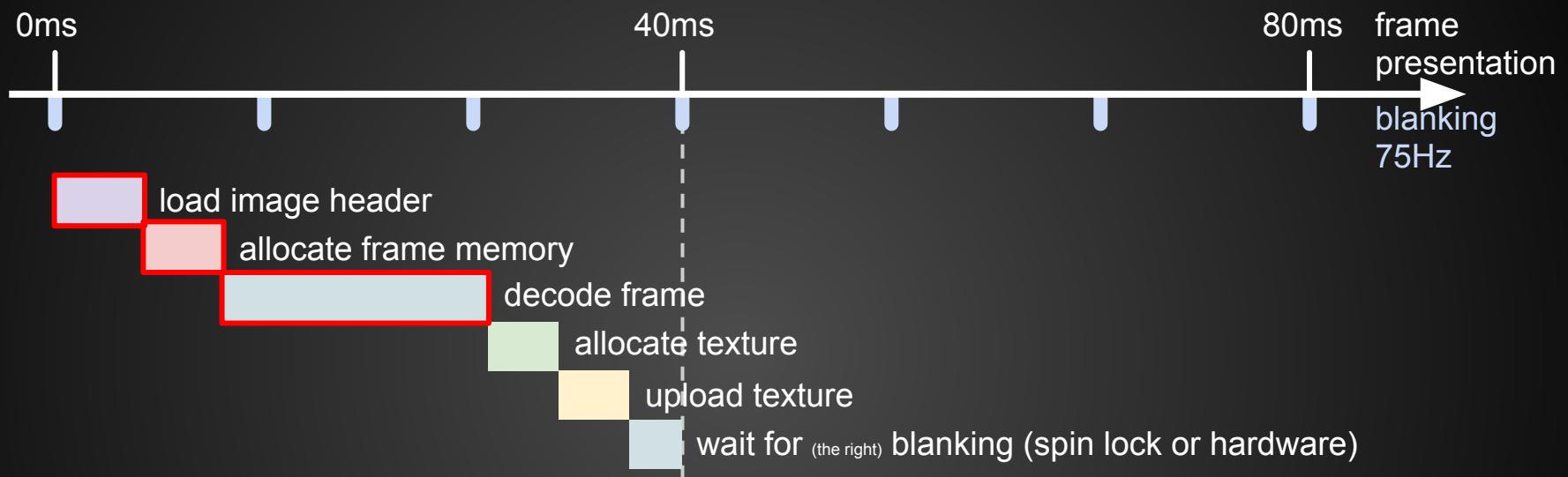
Classical framerates : 24, 25, 29.97, 30...
Classical refreshrates : 50, 60, 72, 75...

25 FPS on 50 Hz display : OK
- 1 frame every 2 blanking

24 FPS on 60 Hz display : KO
- pattern : 2,2,3,...

24 FPS on 50 Hz display : KO
- pattern : 2,2,2,2,2,2,2,2,2,2,2,3,...

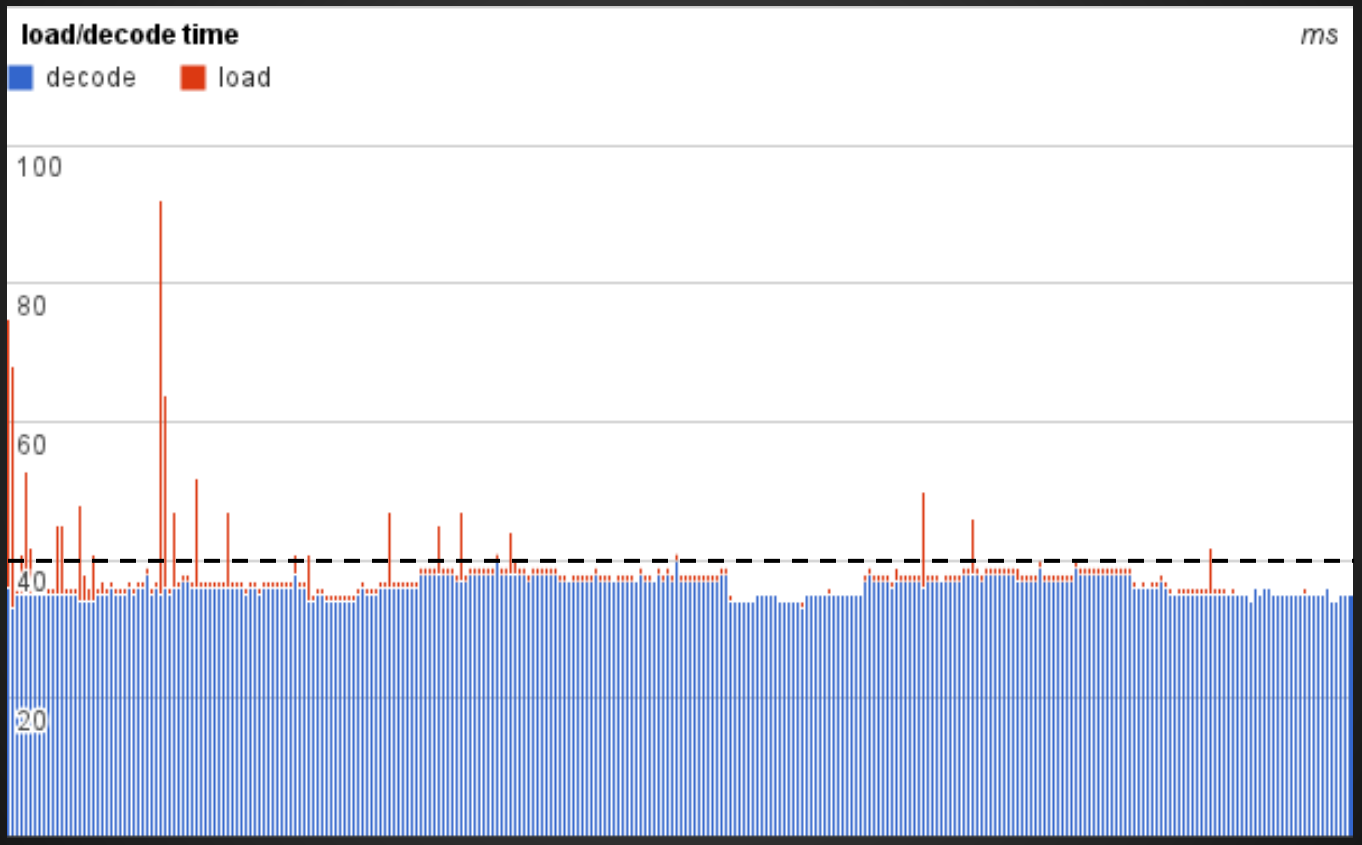
Loading frames, the naive design



Too much uncertainties, time is unbound for

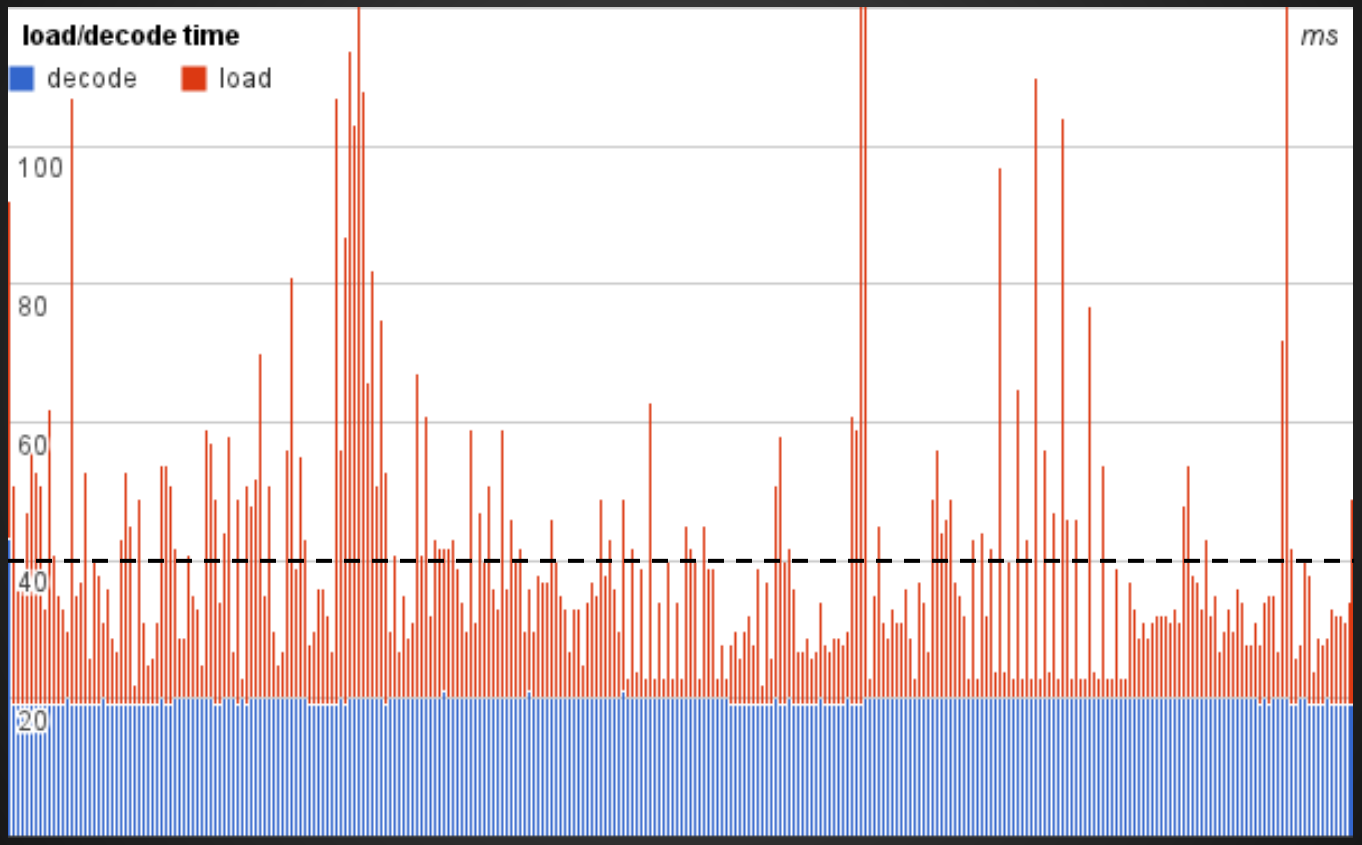
- file reading
  - FS fragmentation, OS caching, permissions checking
- free store allocation
- frame decoding

Impact of frame load/decode time



case 1 linux desktop : jpeg file sequence
bottleneck : decoding, some uncontrolled read peaks

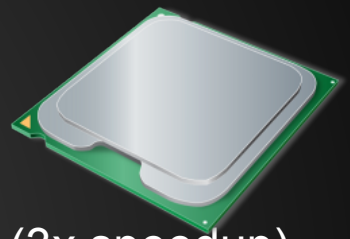
Impact of frame load/decode time



case 2 windows desktop : same jpeg file sequence
bottleneck : fragmented file system, a lot of uncontrolled read peaks

So how to improve?

Do it faster

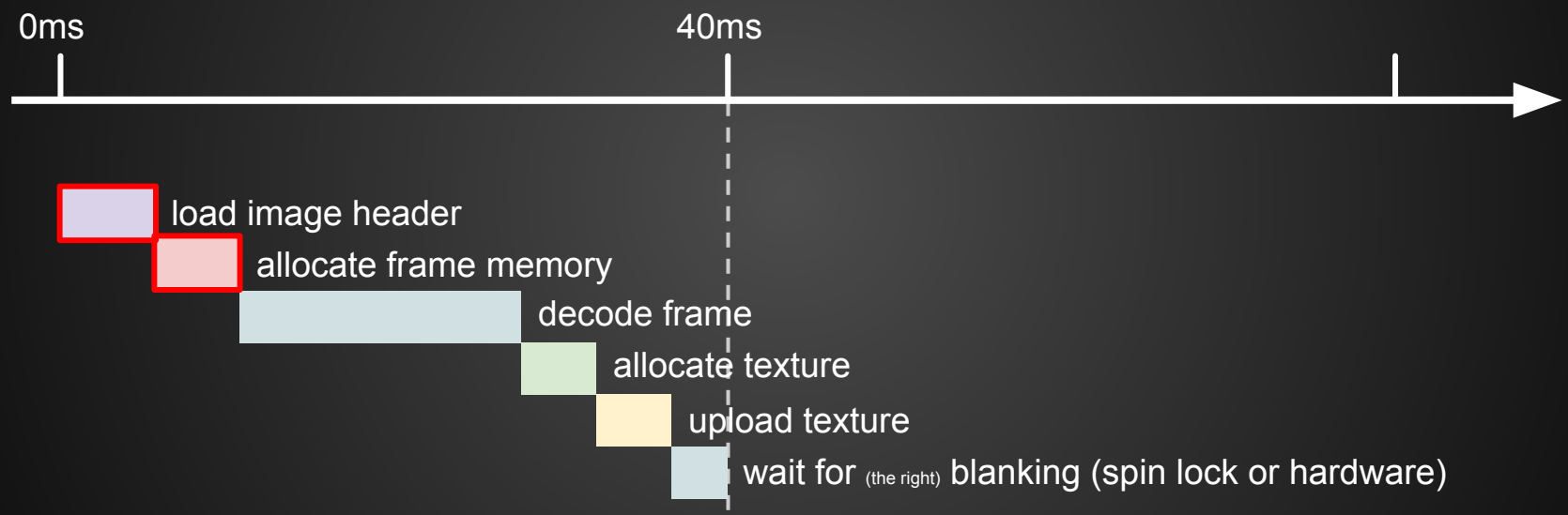


- Improve decoding
  - Choose optimized libraries : libjpeg-turbo (3x speedup)
- Use a cache
  - keep recently decoded frame in memory

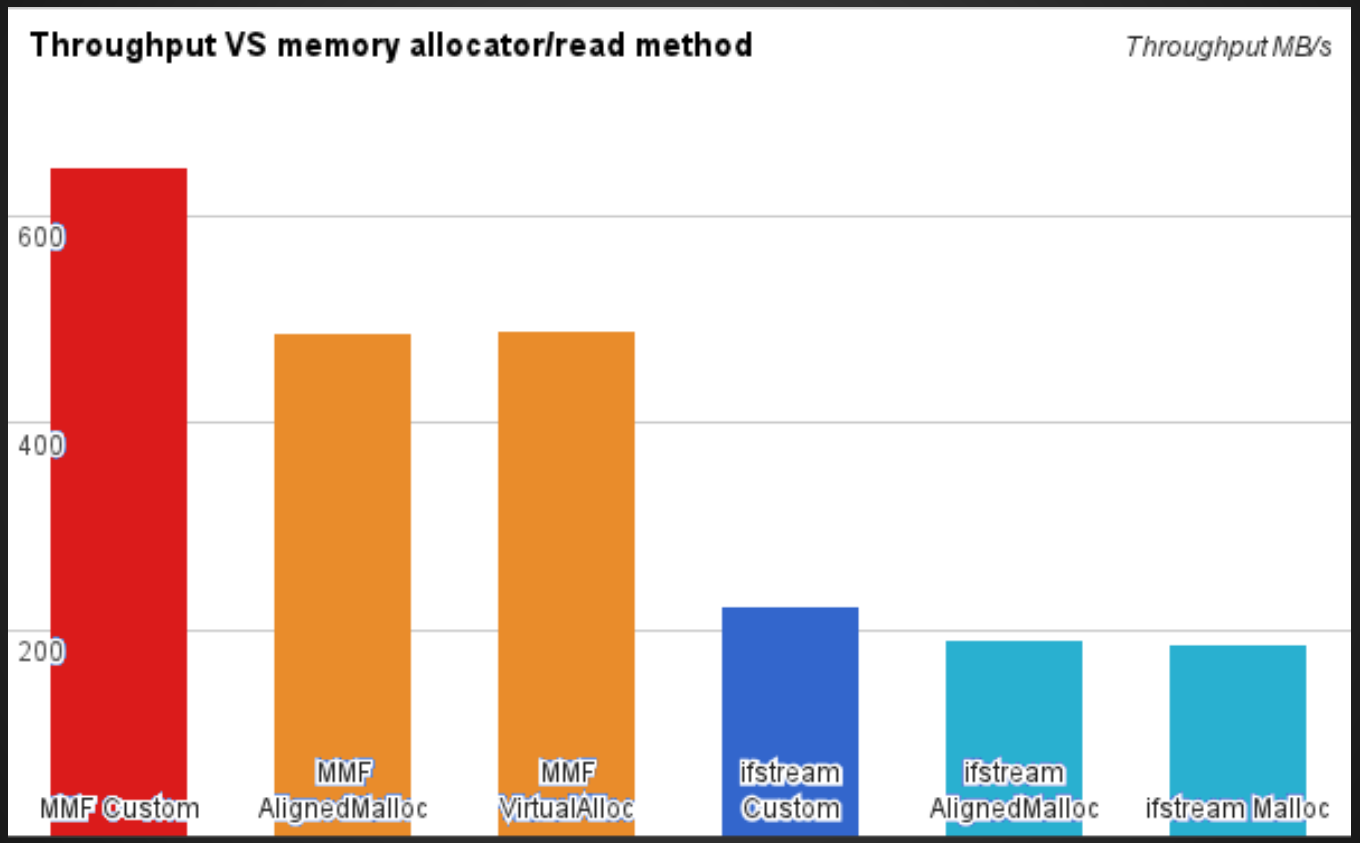
Do it ahead of time "Use the cores Luck"

- Hide latencies by parallelizing
  - load and decode in parallel

Influence of I/O and memory

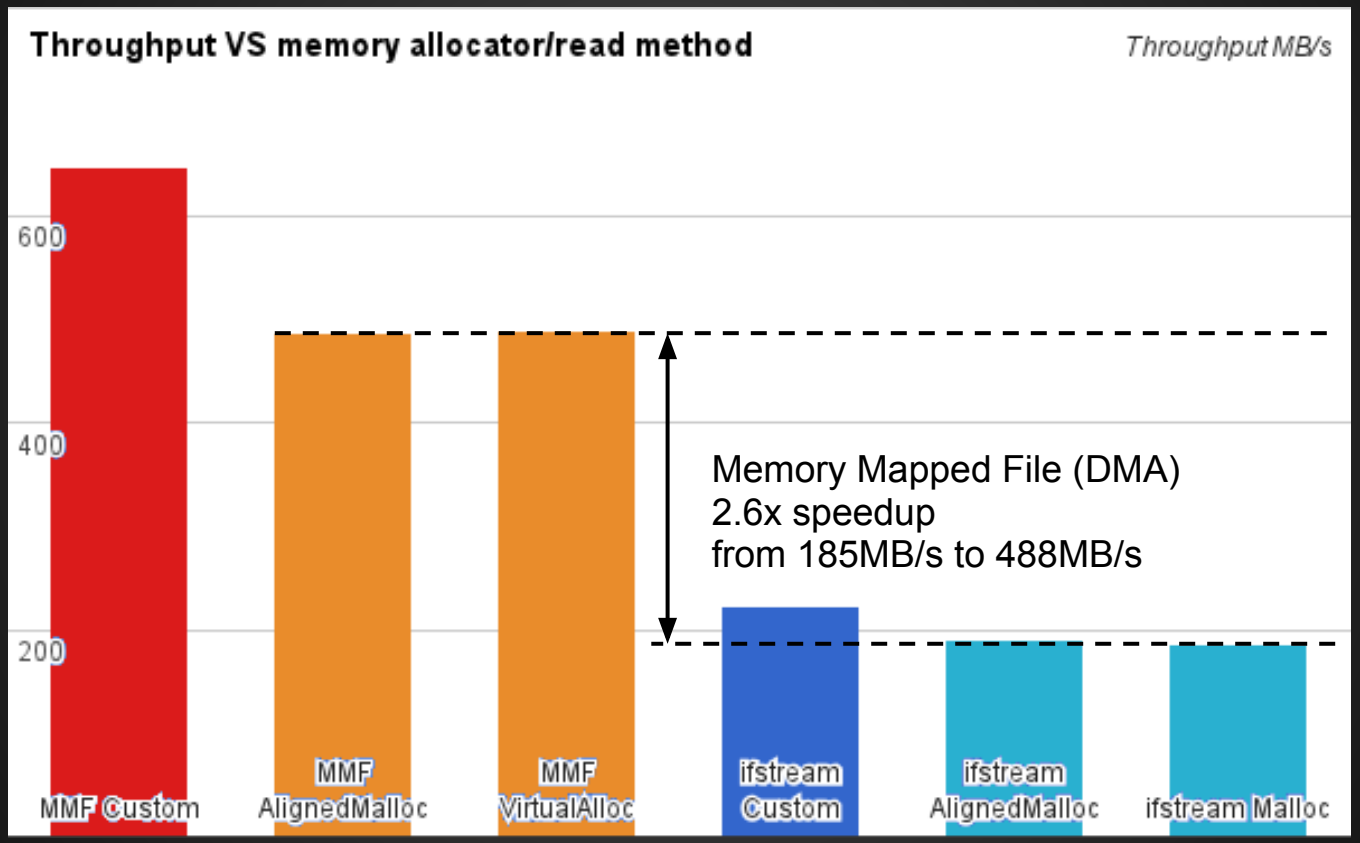


Influence of I/O and memory allocation



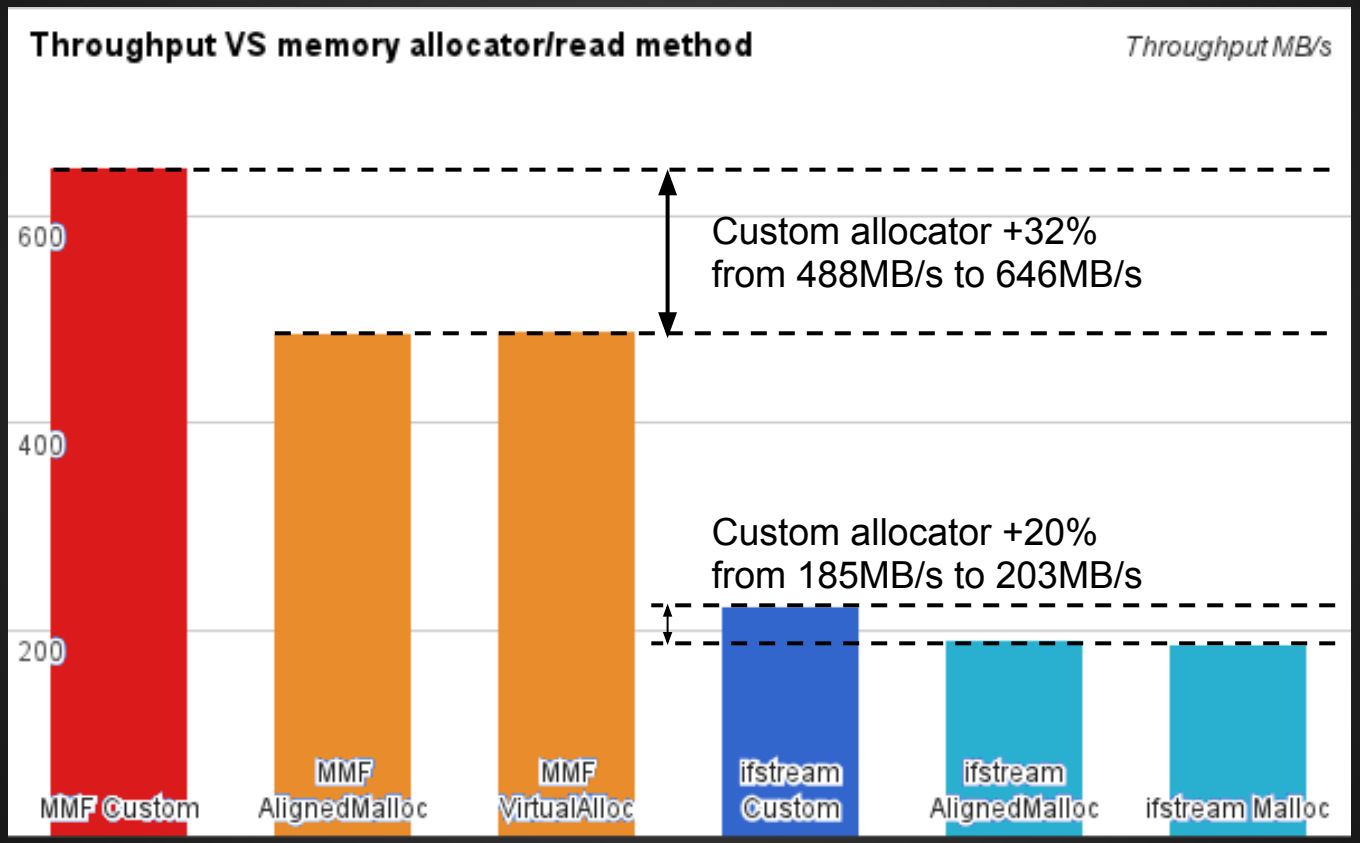
case 3 windows with dedicated high performance storage
dpx file sequence

Influence of I/O and memory allocation



case 3 windows with dedicated high performance storage
dpx file sequence

Influence of I/O and memory allocation



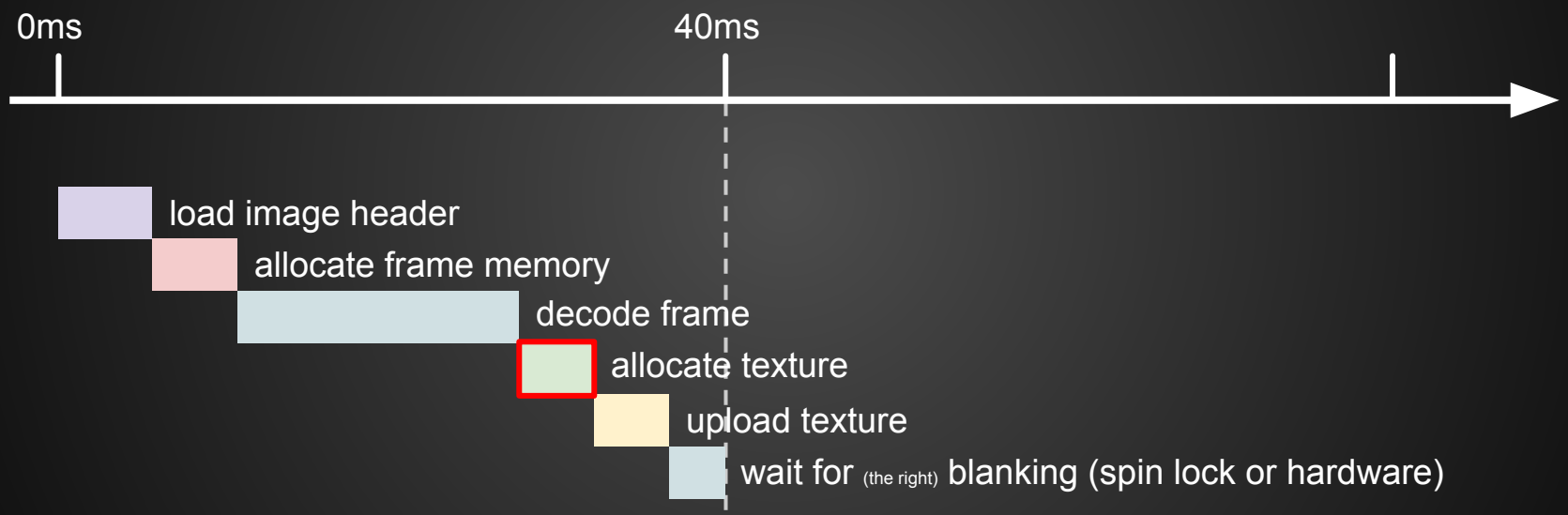
case 3 windows with dedicated high performance storage
dpx file sequence

So how to improve?

Use OS specialized functions

- Bypass OS I/O caches
  - `CreateFile`
  - `FILE_FLAG_NO_BUFFERING | FILE_FLAG_SEQUENTIAL_SCAN`
- Use Memory Mapped Files
  - DMA is your friend
- Create your own allocator if needed
  - When dealing with thousands of MB/s, *malloc* simply isn't fast enough

Optimize even more ?



- Use a texture pool
  - avoid creating new ones each frame

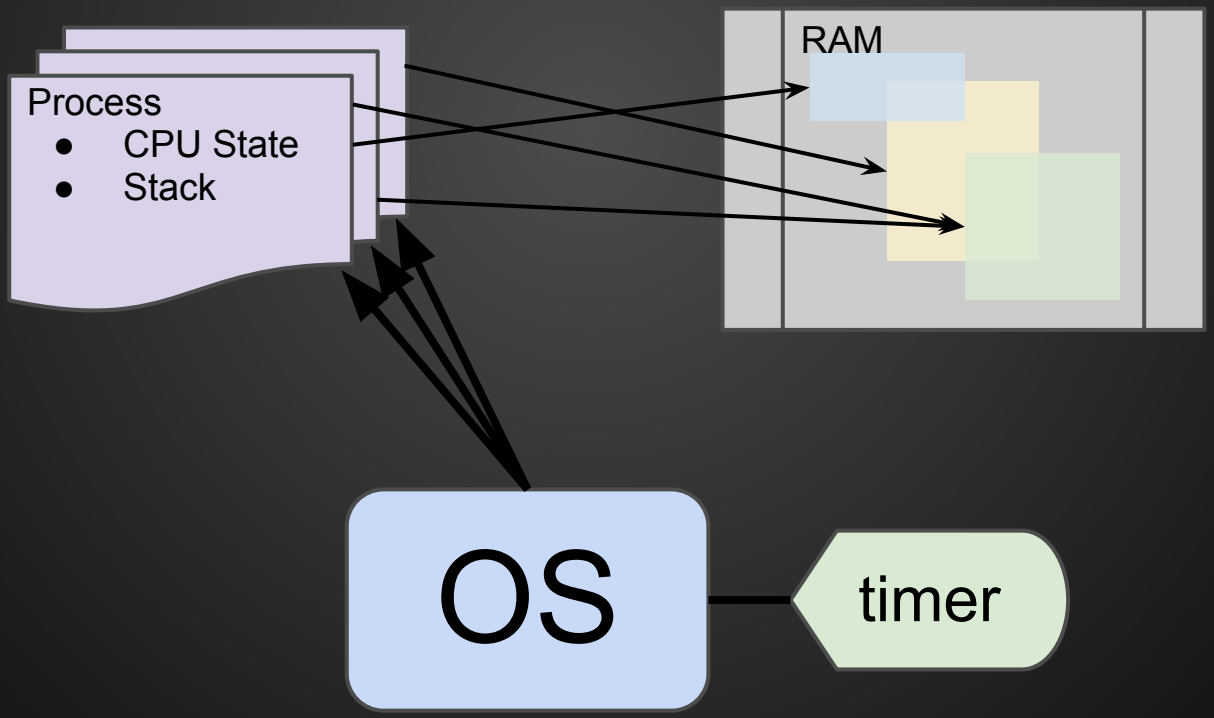
A briefer history of data sharing

Largely inspired by Andrei Alexandrescu's

Concurrency in the D Programming Language

A briefer history of data sharing

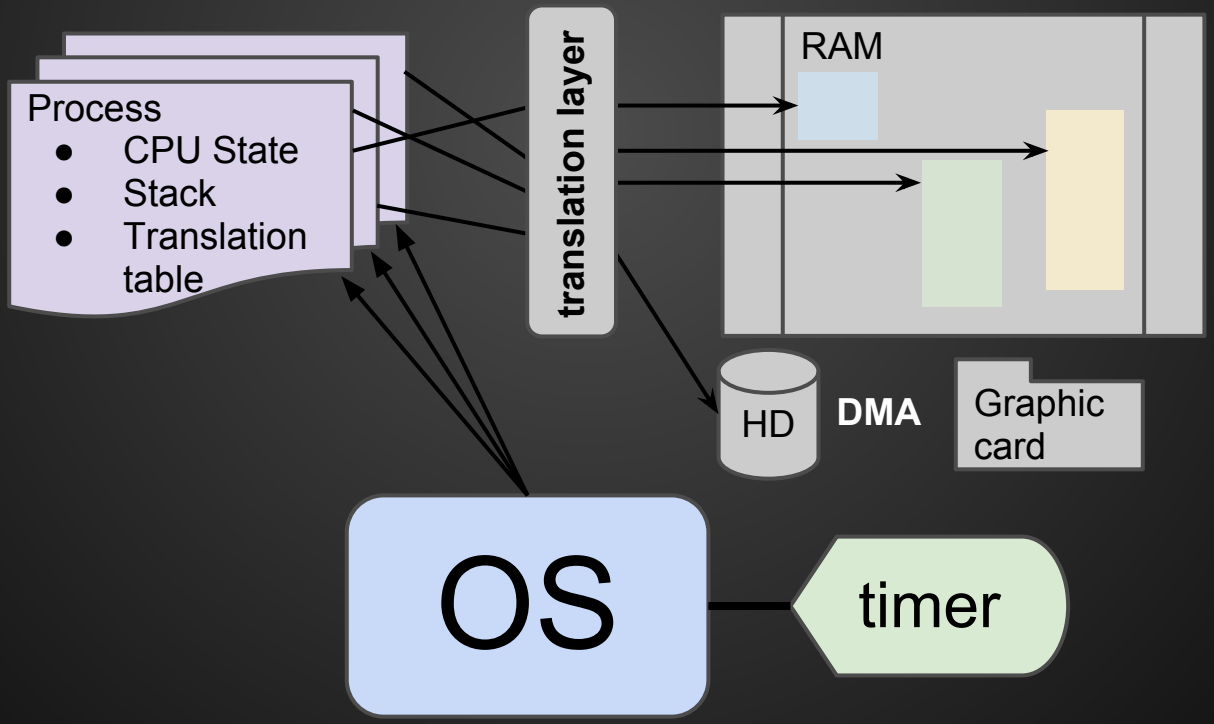
back in the 60's : time sharing



A briefer history of data sharing

memory virtualization => process isolation

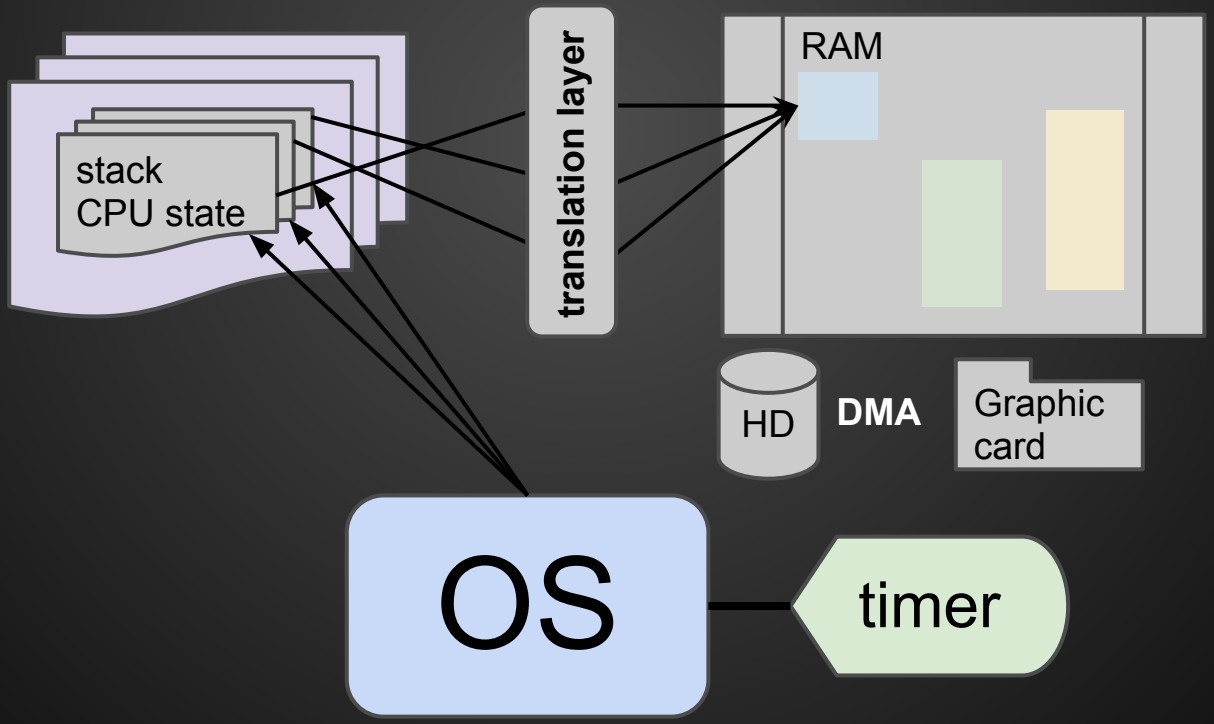
remember DOS unexpected reboot ?



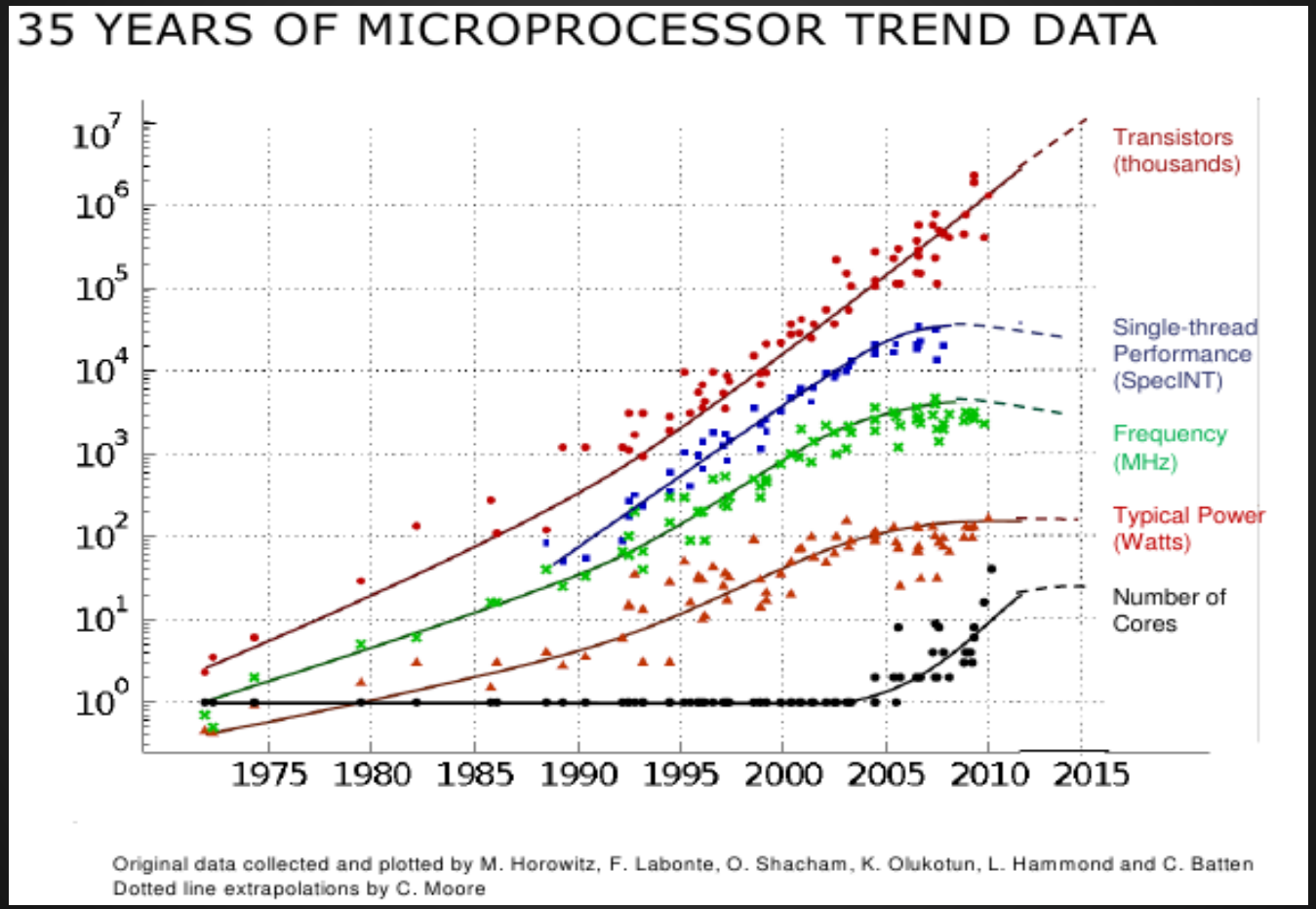
A briefer history of data sharing

thread => bare lightweight process

share the address space of the process



Sky is the limit



from A look back at single threaded performance

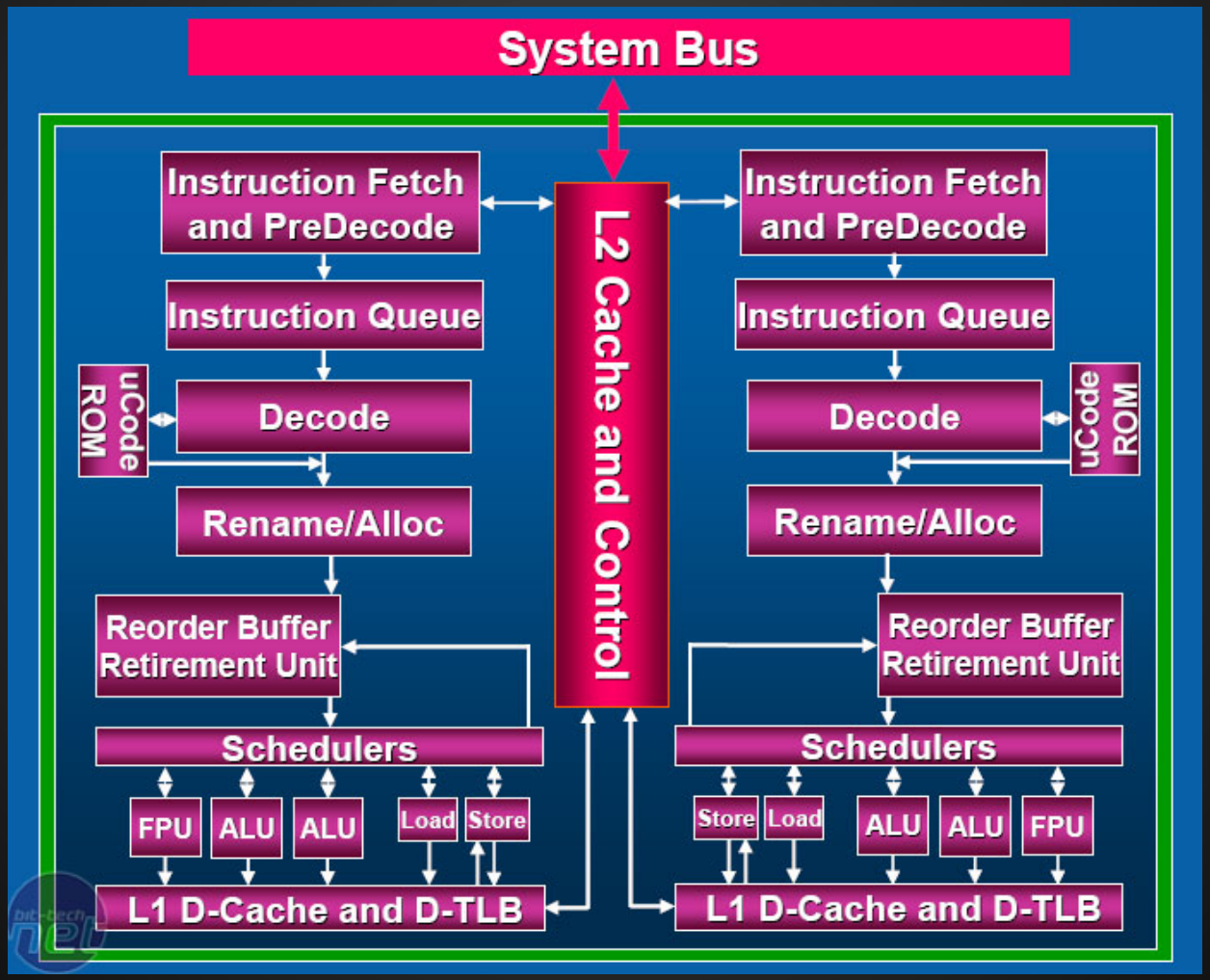
Power is the limit actually

but it's another story...

More on this in this brilliant talk
by Bill Dally Chief Scientist at NVidia

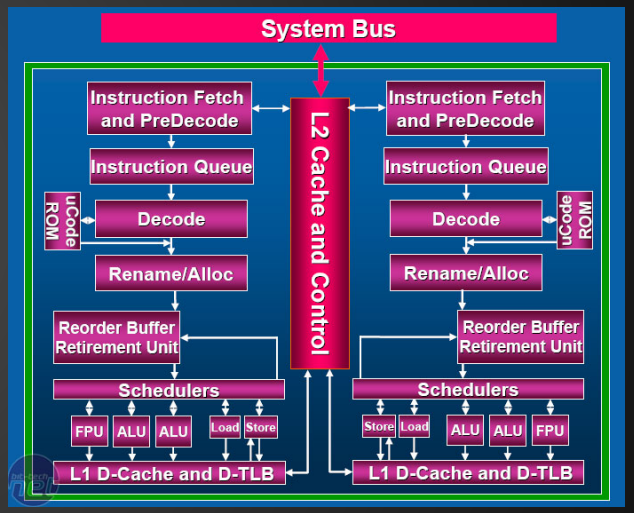
you'll need Silverlight though :(

Modern CPU design implications

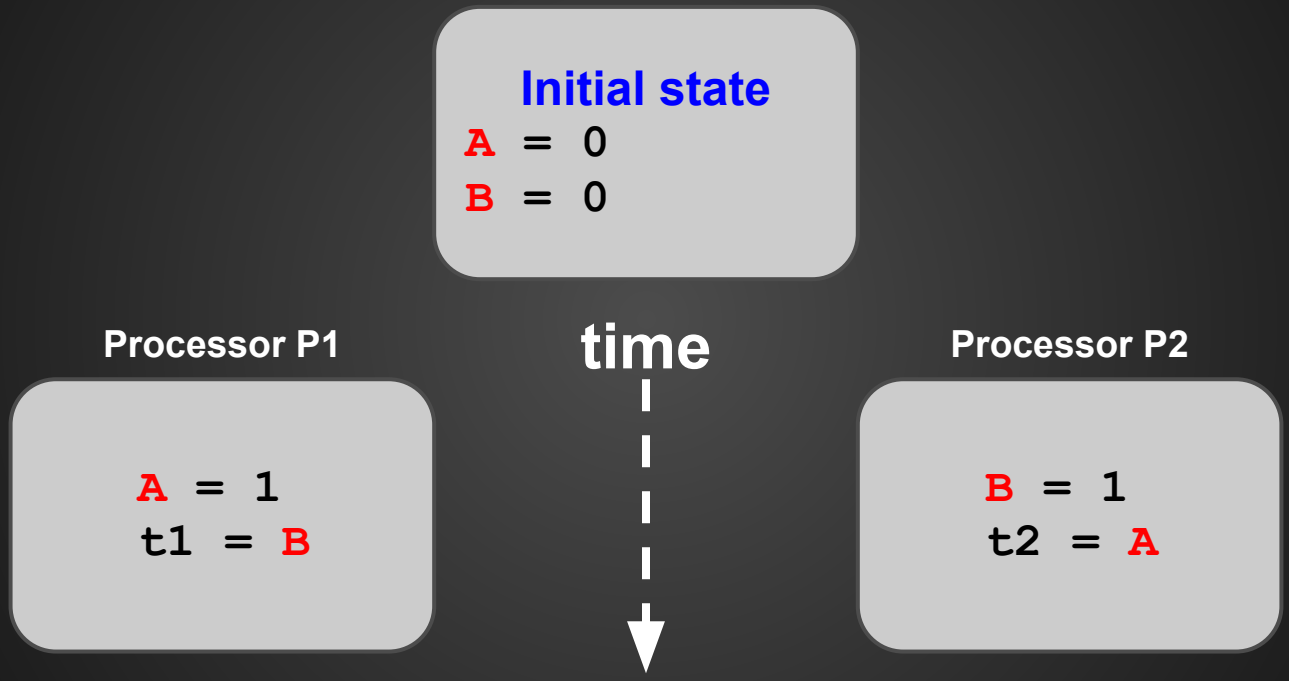


Modern CPU design implications

- CPU frequency >> memory frequency
  - CPU stall, waiting for mem
- cache hierarchy
  - registry, L1, L2, L3, RAM
- CPU features
  - superscalar pipeline
  - out of order execution
- relaxed memory model
  - *happens before relationship* is now explicit
    - use memory barriers
  - Please forget that *volatile* keyword at least for C/C++



The (surprising) x86 memory model



The outcome t1 == 0 and t2 == 0 is possible on the x86

from Understanding Violations of Sequential Consistency
post by **Bartosz Milewski**, credits to **Luis Ceze** from University of Washington

Rule of thumb

All access (write and read) to shared variables **must** be protected with memory barriers

OS provide Semaphore and Mutex via C API
POSIX threads and Windows threads

Compiler vendors provide atomic types, eg CAS

- `__sync_bool_compare_and_swap` (gcc)
- `InterlockedCompareExchange` (visual)

Multi-thread coding is hard... seriously

threads communication **bite** beware of

deadlocks and starvation

Debugging is super hard

- every execution is different
- load sensitive (debug & release)
- mixed output

Use higher level primitives

- Boost Thread
- TBB (Intel threading building blocks)
- Don't share, if you can afford copying around
  - Message Passing a la Erlang...

Back to our cachelookaheadthingy

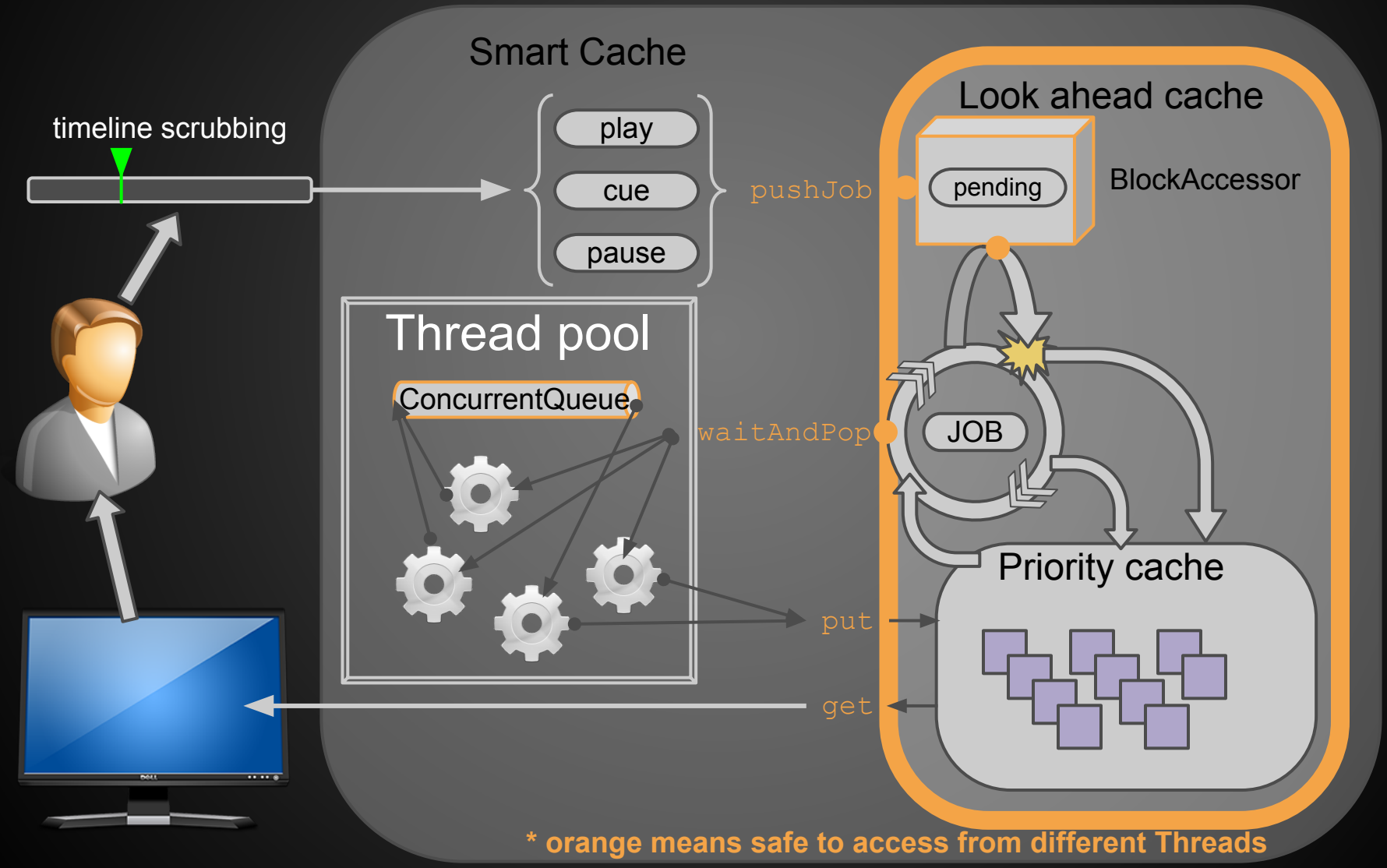
We aim at perfect playback, so :

- load images fast
- scrub the timeline smoothly
- prefetch according to playback
  - both ways when paused, forward or reverse
- cache for reuse

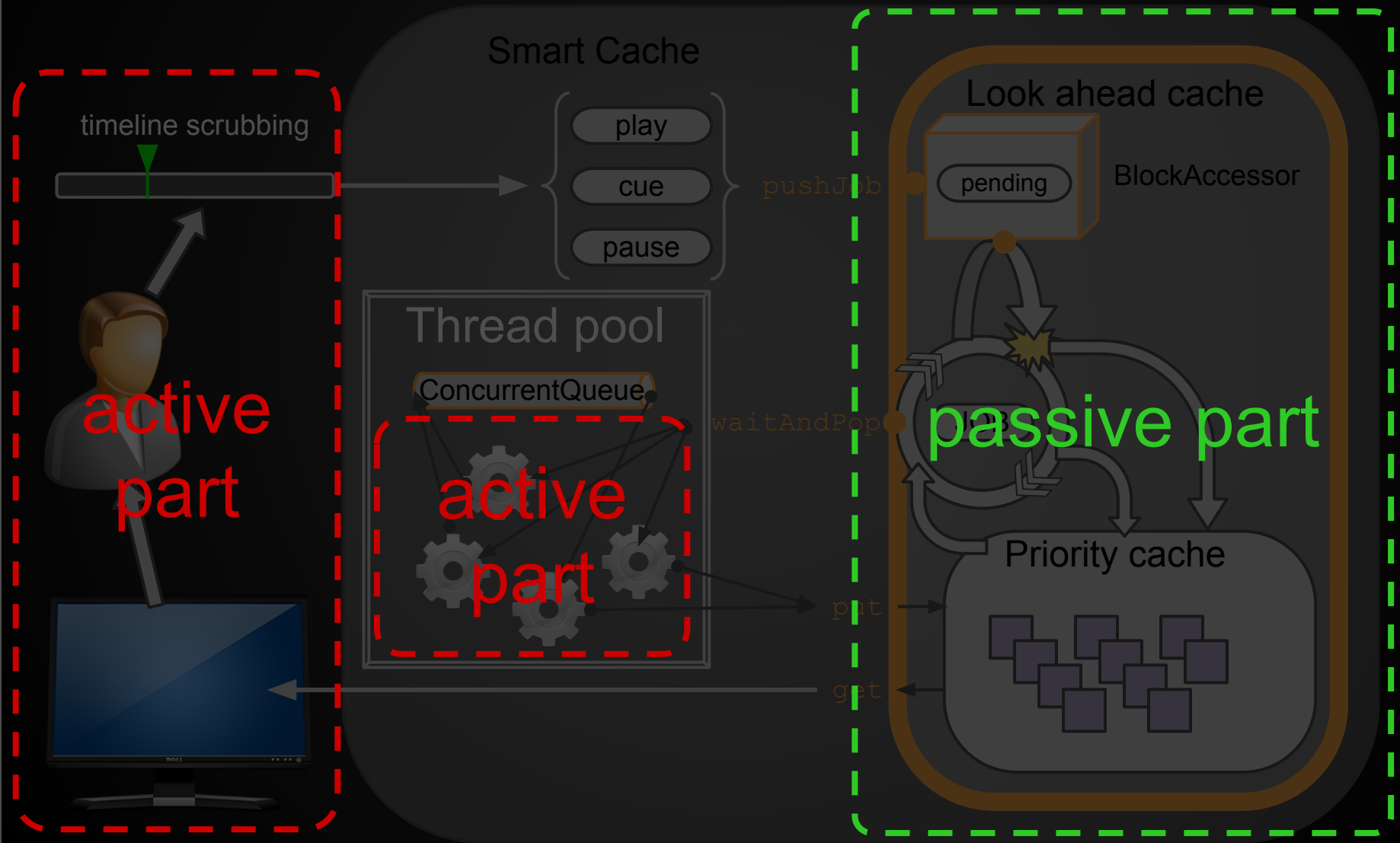
My last 7 previous designs

- Can't scale with the machine
  - Load/decode up to X predetermined frames
    - we might have GBs of RAM
  - Max of 2 threads : 1 decode & 1 load
    - we might have 16 cores
  - Only one file per frame
    - we might have several tracks or stereo
  - More threads would mean more contention
- Highly specialized => not reusable
- Too complicated and buggy

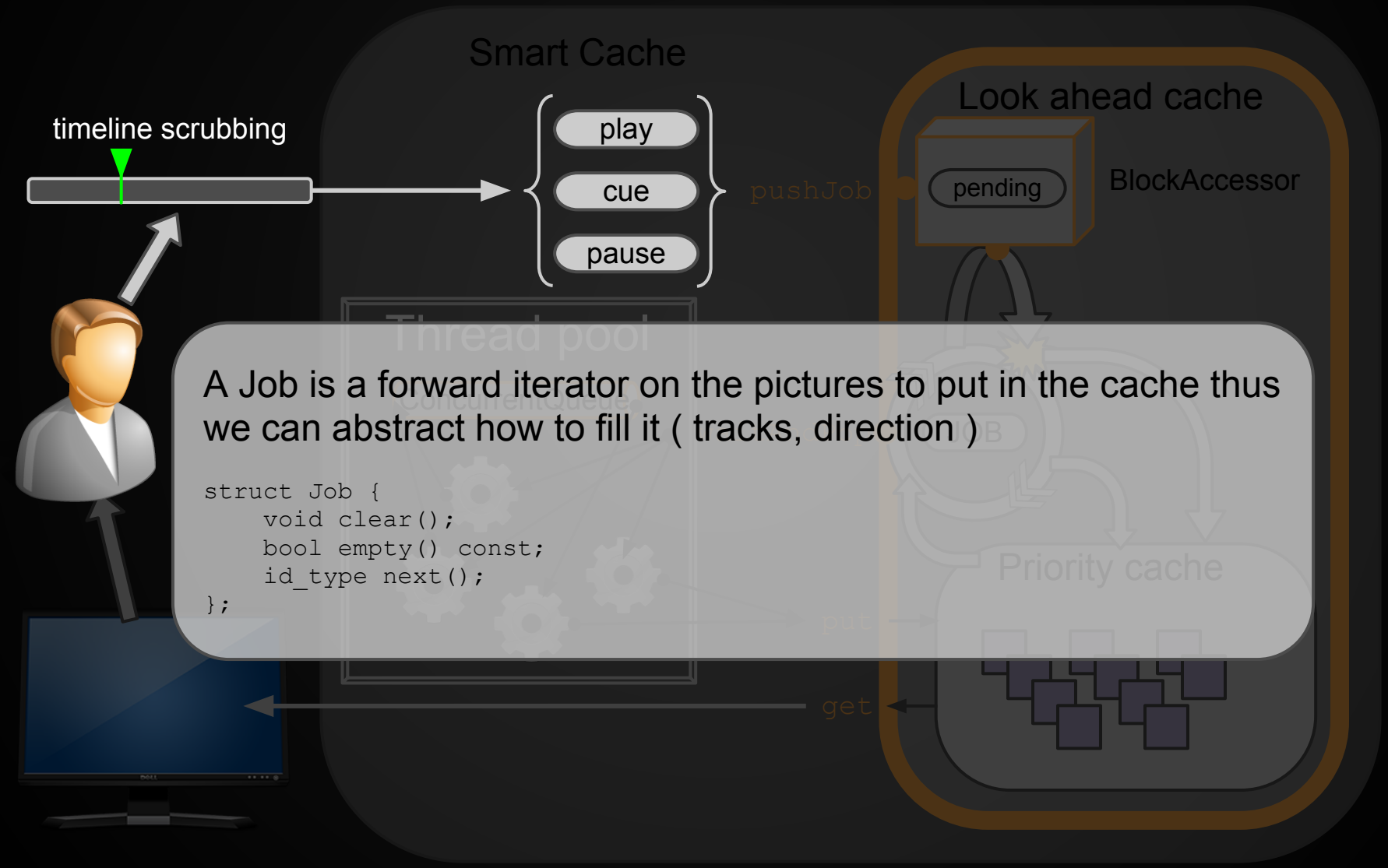
The new design



The new design



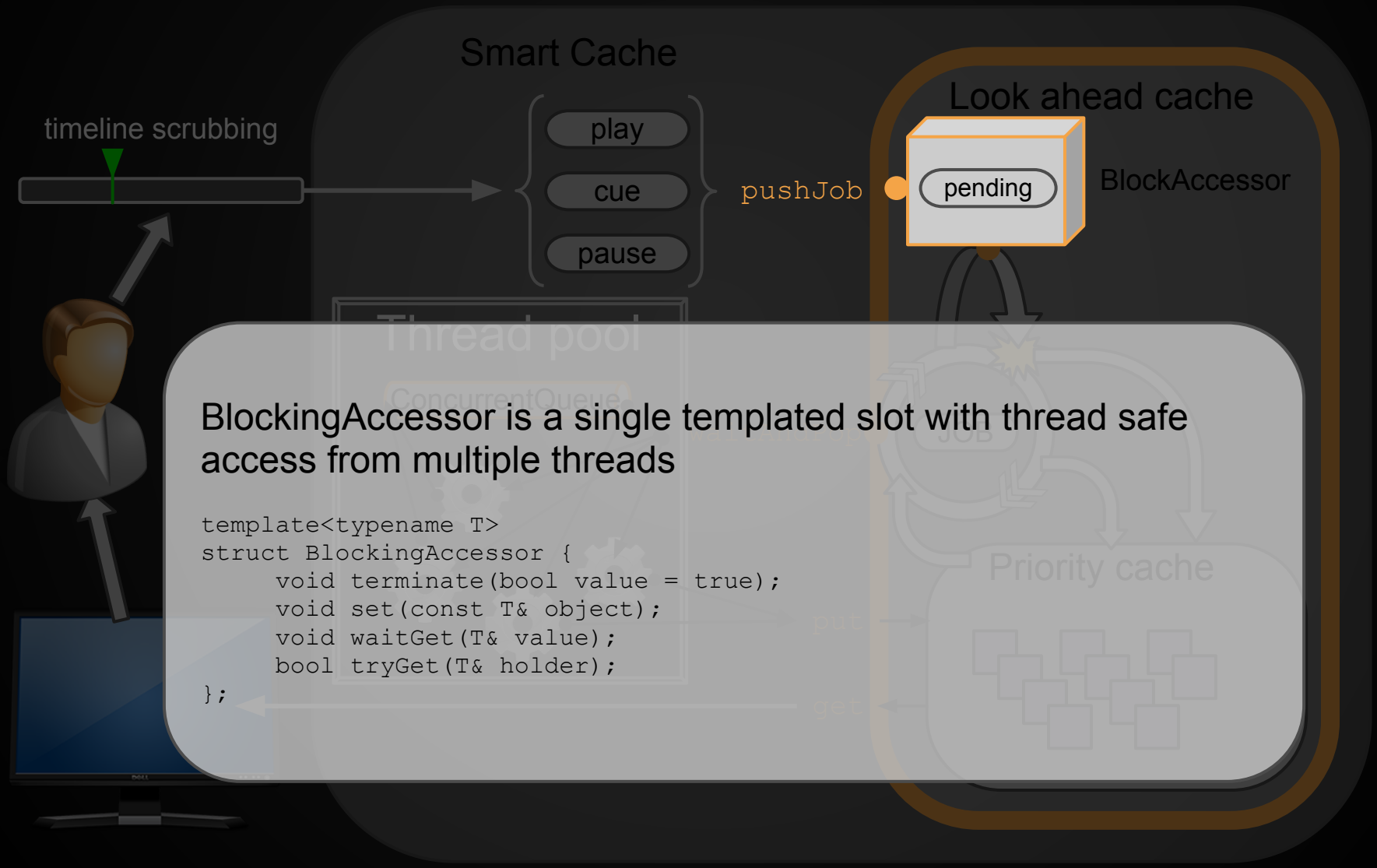
The new design



A Job is a forward iterator on the pictures to put in the cache thus we can abstract how to fill it (tracks, direction)

```
struct Job {
    void clear();
    bool empty() const;
    id_type next();
};
```

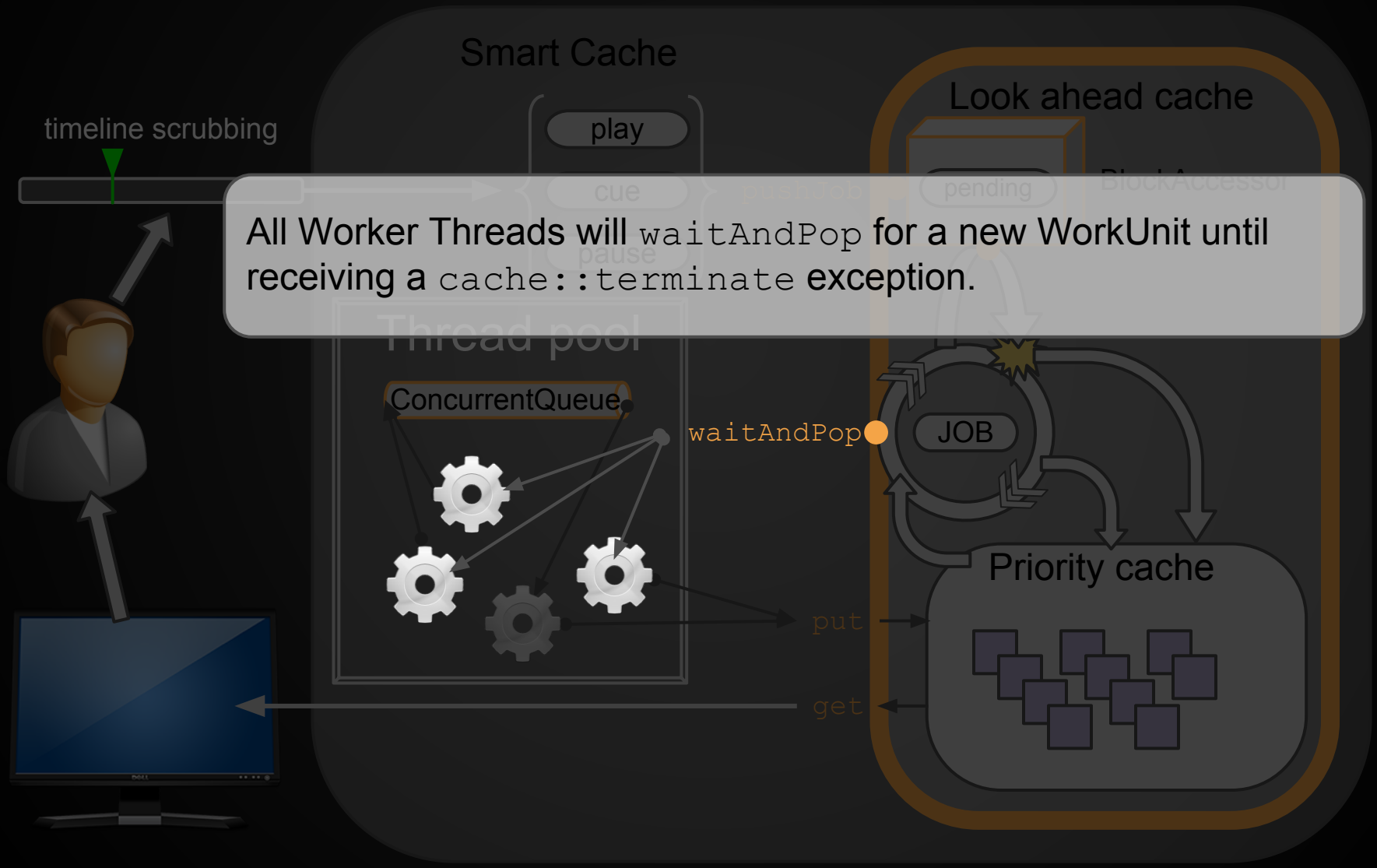
The new design



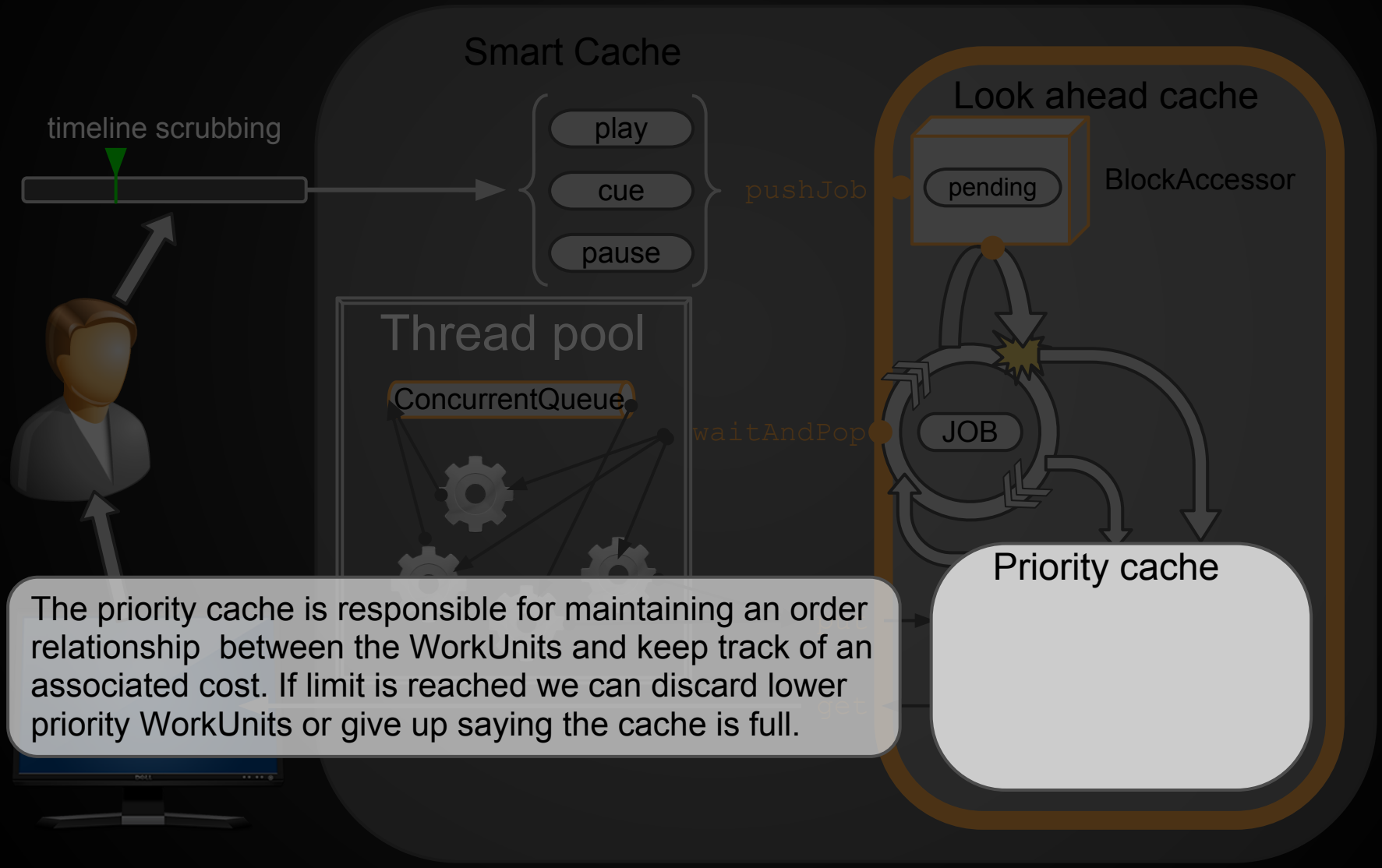
BlockingAccessor is a single templated slot with thread safe access from multiple threads

```
template<typename T>
struct BlockingAccessor {
     void terminate(bool value = true);
     void set(const T& object);
     void waitGet(T& value);
     bool tryGet(T& holder);
};
```

The new design



The new design

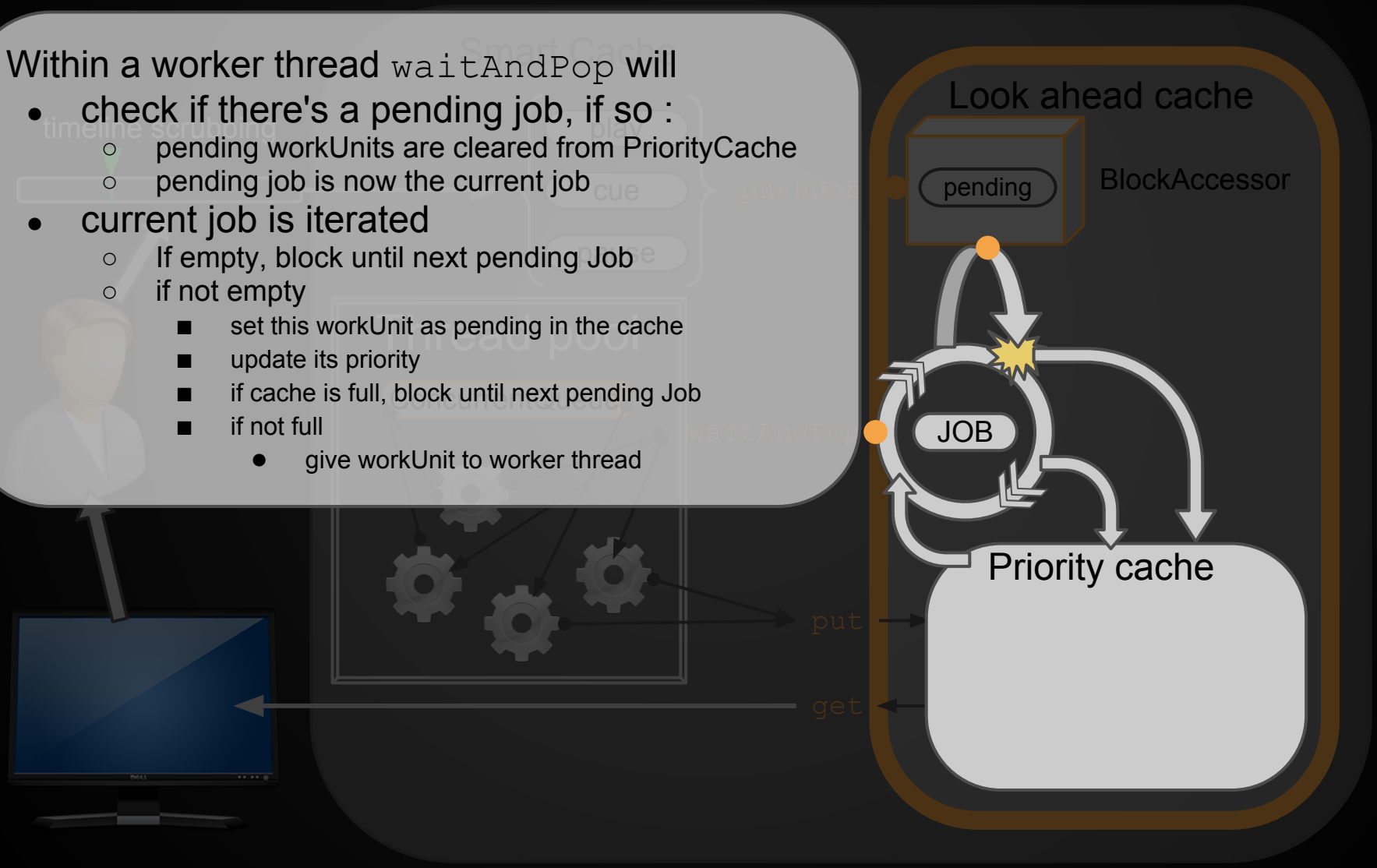


The priority cache is responsible for maintaining an order relationship between the WorkUnits and keep track of an associated cost. If limit is reached we can discard lower priority WorkUnits or give up saying the cache is full.

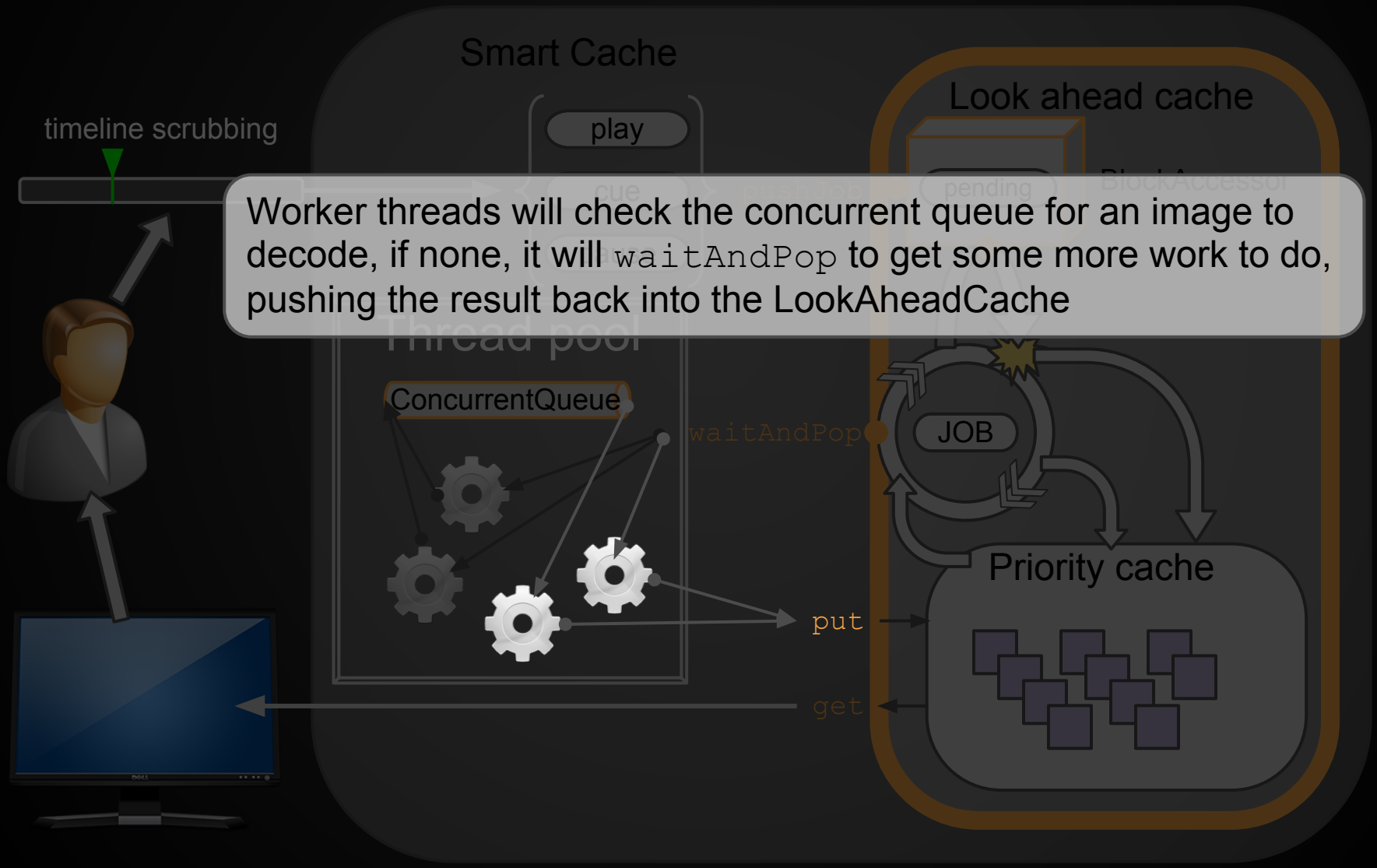
The new design

Within a worker thread `waitAndPop` will

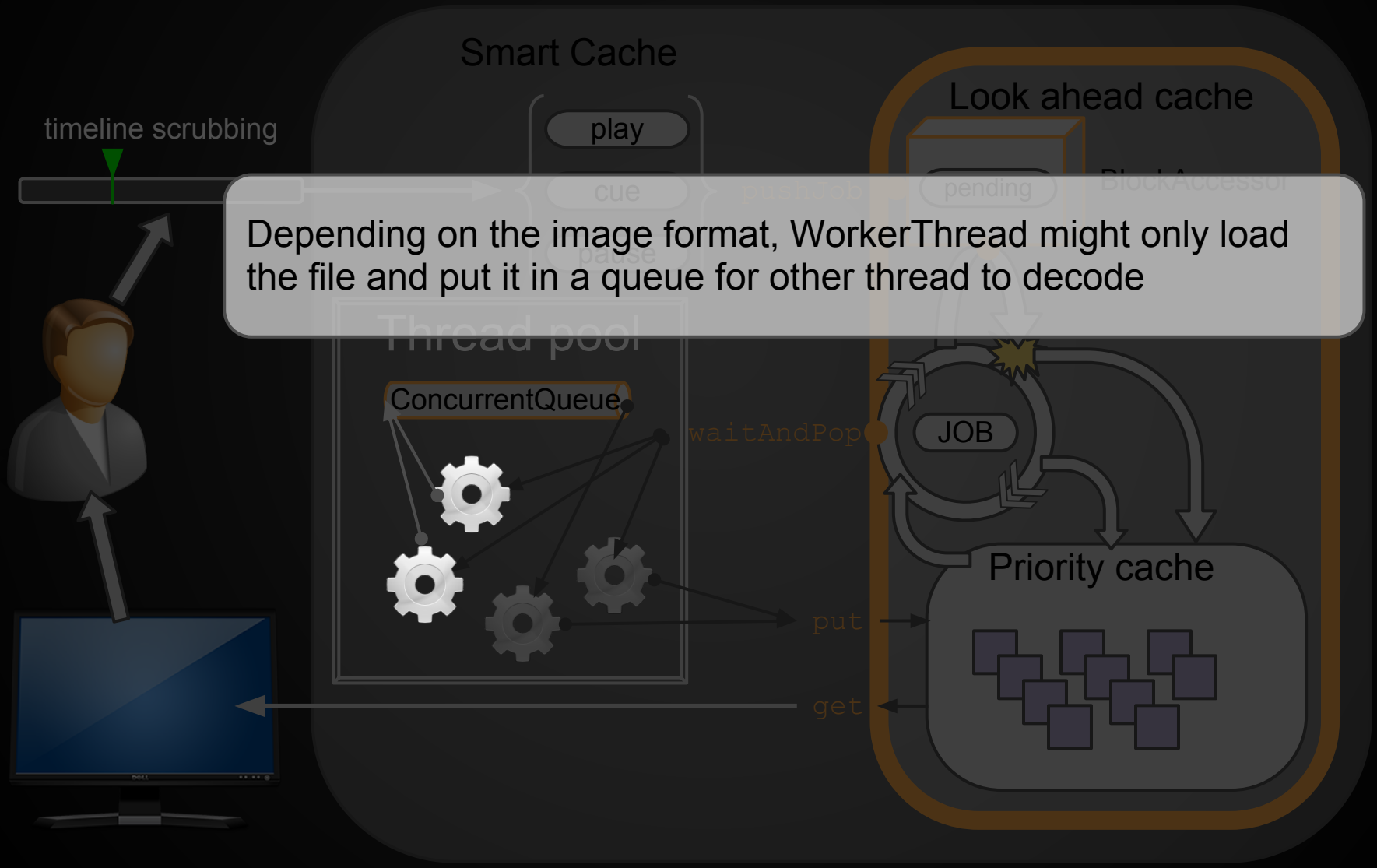
- check if there's a pending job, if so :
  - pending workUnits are cleared from PriorityCache
  - pending job is now the current job
- current job is iterated
  - If empty, block until next pending Job
  - if not empty
    - set this workUnit as pending in the cache
    - update its priority
    - if cache is full, block until next pending Job
    - if not full
      - give workUnit to worker thread



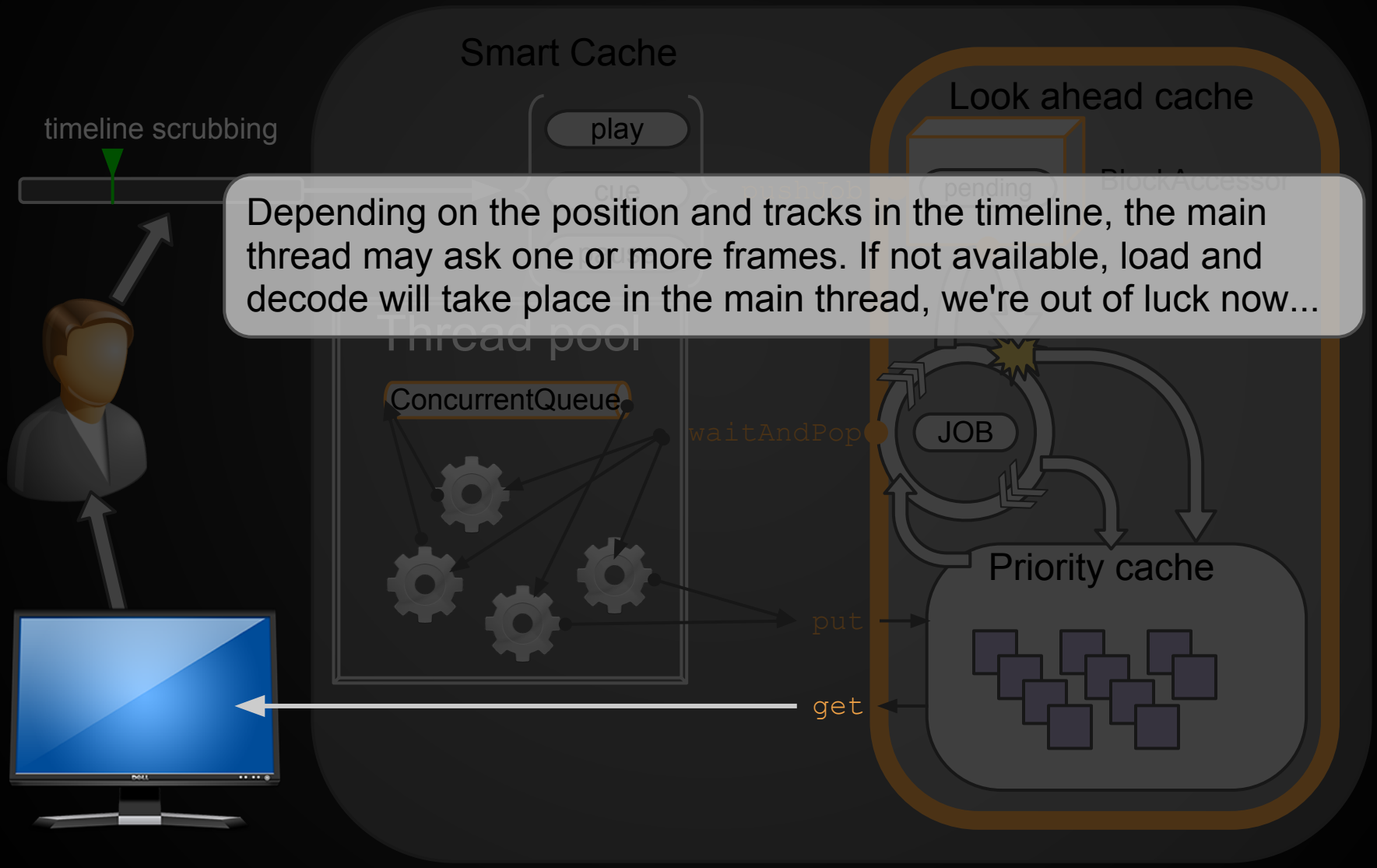
The new design



The new design



The new design

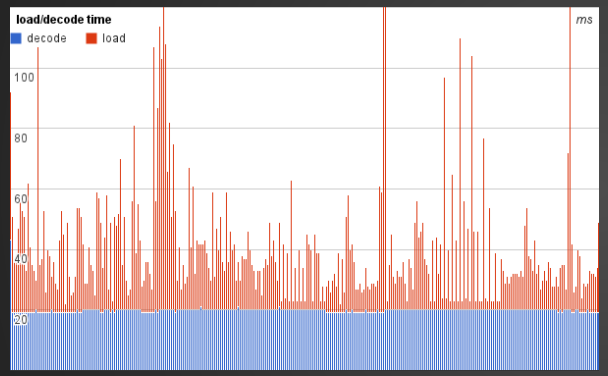


Demo

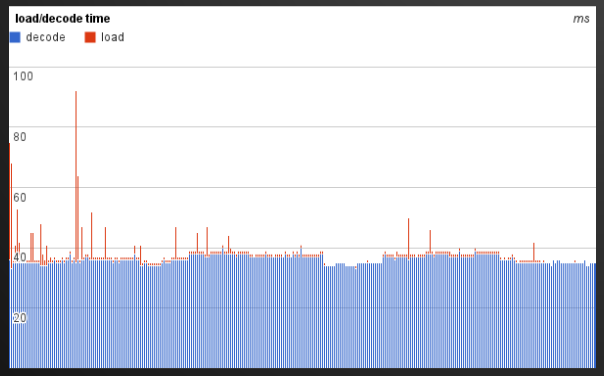
let's rock and roll !

So, how does it perform ?

Remember those ?

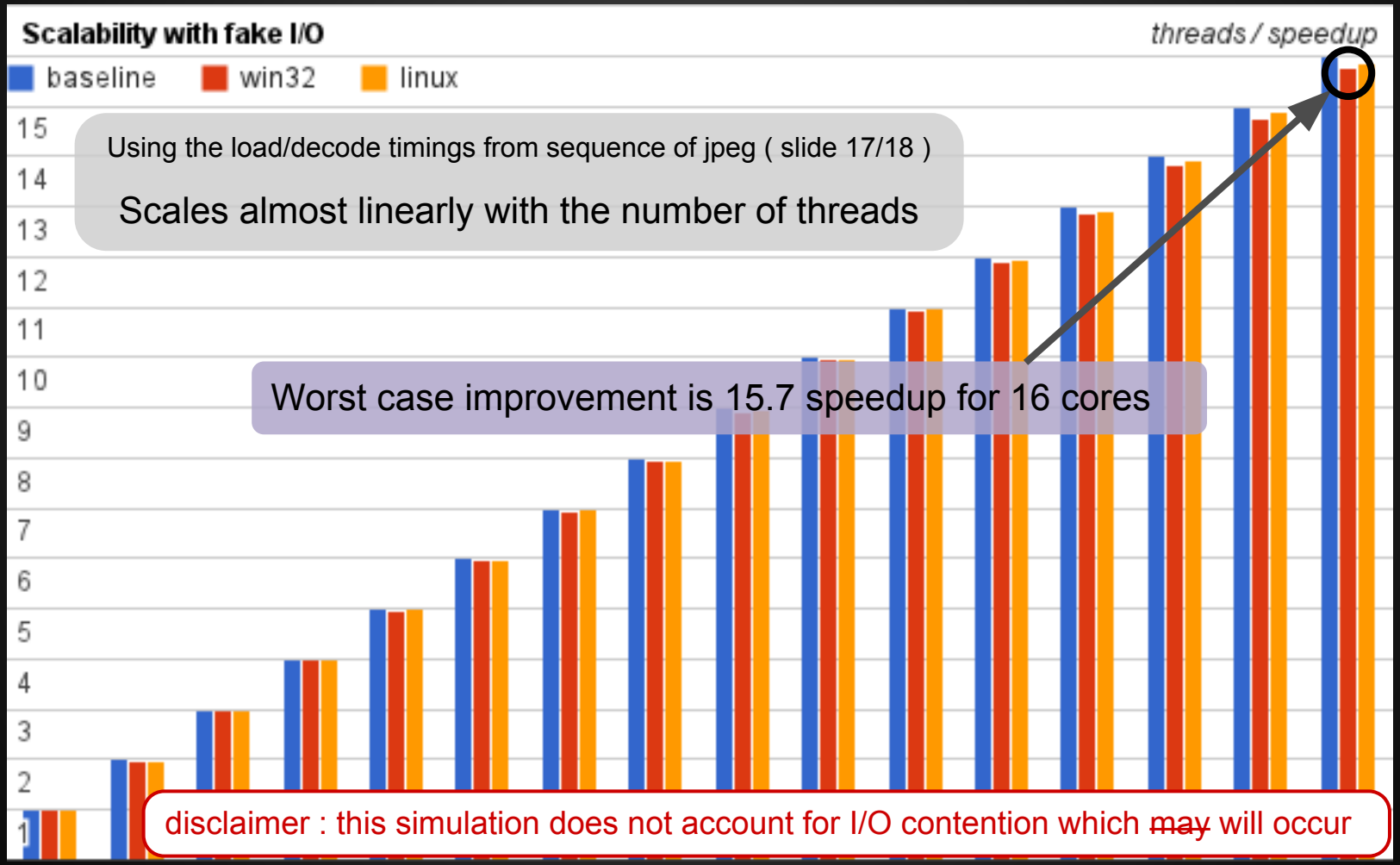


jpeg sequence on windows



jpeg sequence on linux

So, how does it perform ?



Conclusions

Look Ahead Cache

- good scalability (small contention)
- still need some work
- reusable in different contexts

Library design, one of the noblest goal for a developer

- hard but pays off in the long run
- takes time to find a good design

Duke

- Open Source, free, multiplatform (lin, win, mac)
- binaries to be released (lin/win for the moment)
- much, much more to say about ...

Q&A

Thanks for your attention :)