Prudent Memory Reclamation in Procrastination-Based Synchronization

Aravinda Prasad & K Gopinath

Computer Science & Automation (CSA), Indian Institute of Science (IISc), Bangalore

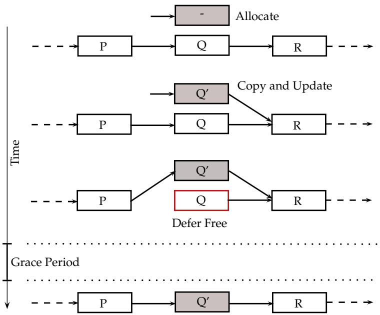
{aravinda, gopi}@csa.iisc.ernet.in

EECS 2016

Synchronization via Procrastination

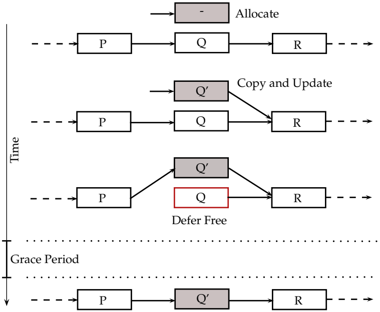
- Readers:
  - Do not synchronize with writers
  - Are wait free and scale linearly
- Writers:
  - Copy the object and update the copied version
  - Wait for pre-existing readers referring the old version to complete

Example: Read-Copy-Update (RCU)



Synchronization via Procrastination

- Readers:
  - Do not synchronize with writers
  - Are wait free and scale linearly
- Writers:
  - Copy the object and update the copied version
  - Wait for pre-existing readers referring the old version to complete



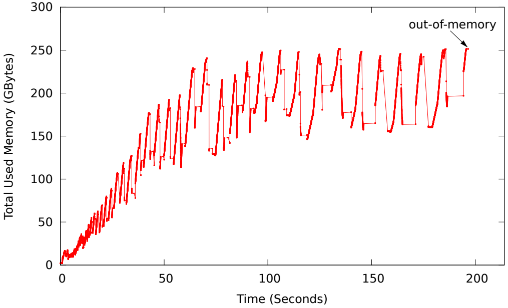
Example: Read-Copy-Update (RCU)

A deferred object is safe for freeing after the completion of all pre-existing readers

Synchronization via Procrastination from Memory Allocator's Point of View

- Frequent allocation and freeing of objects
- Object allocation is spread over an interval of time. Freeing occurs in bursts
- Reclamation of safe deferred objects is
  - Controlled by synchronization mechanism
  - Oblivious of the memory allocator state

Impact of RCU on the SLUB[1] allocator in the Linux kernel



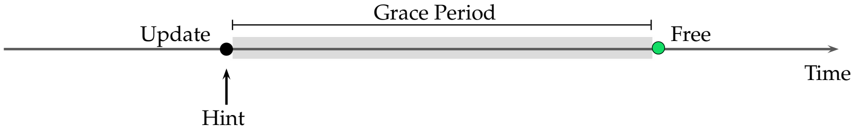
Test Setup:

- Intel Xeon processor with 64 CPUs (4 sockets, 8 cores/socket, 2-way HT)
- 252 GB physical memory. Linux 3.17 kernel.
- Workload continuously performs update operation on 512 byte objects

[1]SLUB is the recent allocator in the Linux kernel based on the slab allocator

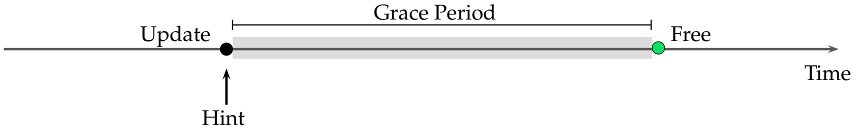
Hints about the future

Deferred frees provide "precise hints" about the memory regions that are about to be freed



Hints about the future

Deferred frees provide “precise hints” about the memory regions that are about to be freed



Given hints about the future, can dynamic memory allocators perform better?

The Prudence Dynamic Memory Allocator

The basic design principle is to have deferred objects visible and processed in the memory allocator

The Prudence Dynamic Memory Allocator

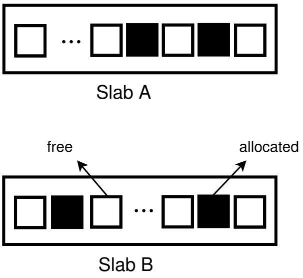
The basic design principle is to have deferred objects visible and processed in the memory allocator

Requirement		**Solution**
Interface to defer the freeing of an object	⇒	Export a new API to defer free an object (`free_deferred()`)
Identify safe time to reclaim deferred objects	⇒	Integrate synchronization mechanism with Prudence

Exploiting hints in Prudence

Reducing total fragmentation with hints

$$\text{Total fragmentation} = \frac{allocated}{requested} = \frac{slabs\_allocated \times slab\_size}{objects\_requested \times object\_size}$$

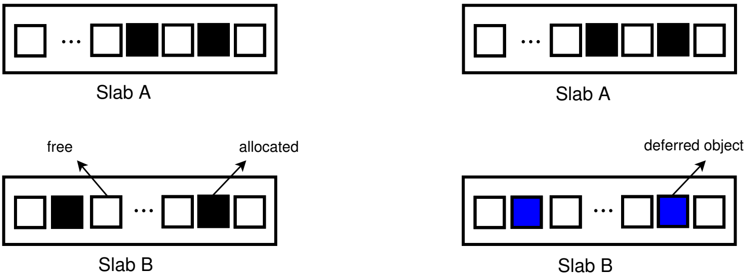


(a) Without Hints

Exploiting hints in Prudence

Reducing total fragmentation with hints

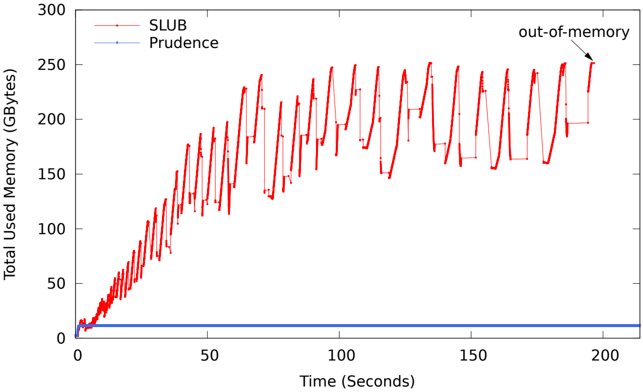
$$\text{Total fragmentation} = \frac{\textit{allocated}}{\textit{requested}} = \frac{\textit{slabs\_allocated} \times \textit{slab\_size}}{\textit{objects\_requested} \times \textit{object\_size}}$$



(a) Without Hints **(b) With Hints**

Prudence selects slab A

Results - Endurance



Prudence does not suffer from high slab churns

Summary

Synchronization via procrastination has direct impact on the performance of memory allocators

★ *Performance impact can be avoided by having deferred objects visible to memory allocators*

Deferred frees provide hints about the memory regions that are about to be freed

★ *Optimizations based on hints can be exploited to improve the performance of memory allocators*

Questions?

Reference:

"Prudent Memory Reclamation in Procrastination-Based Synchronization", Aravinda Prasad, K Gopinath, ASPLOS 2016

Thank you!!