

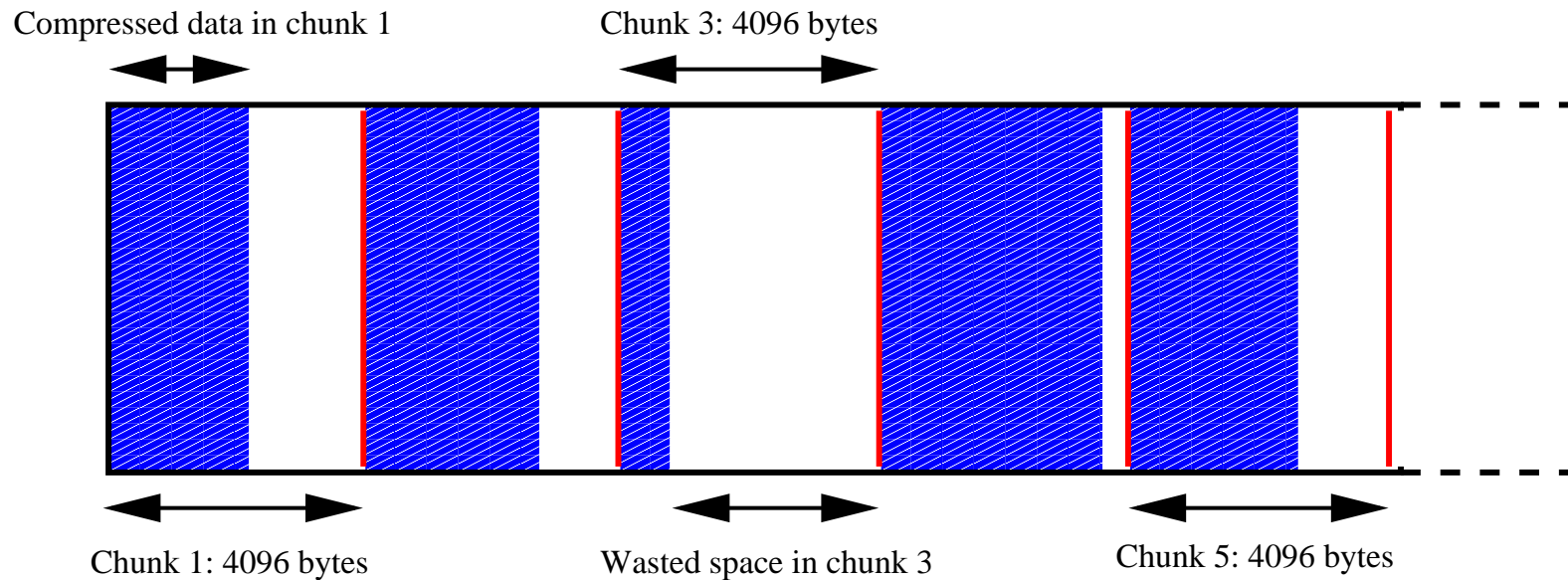


Linux Lab 2010 - Workshop 5

Richard Hayden `rh@doc`

Introduction

Part 5: Claiming back the space we've saved due to compression . . .

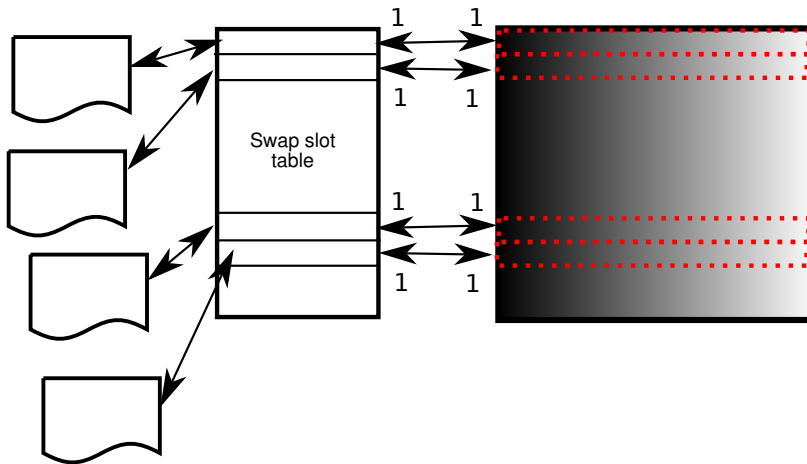


Introduction

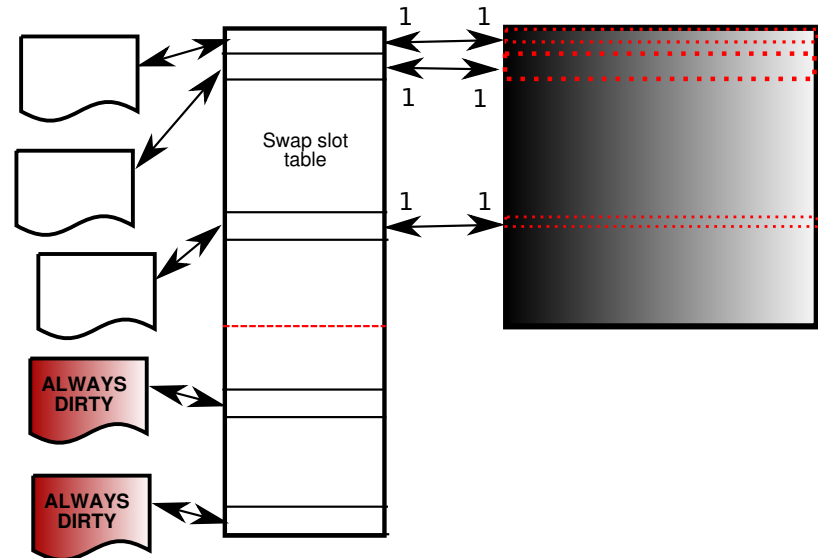
- ➔ Many ways to do this
- ➔ Each will have many parameters and associated trade-offs you could investigate
- ➔ To pass assessment, just have to do reasonably well

Before and after

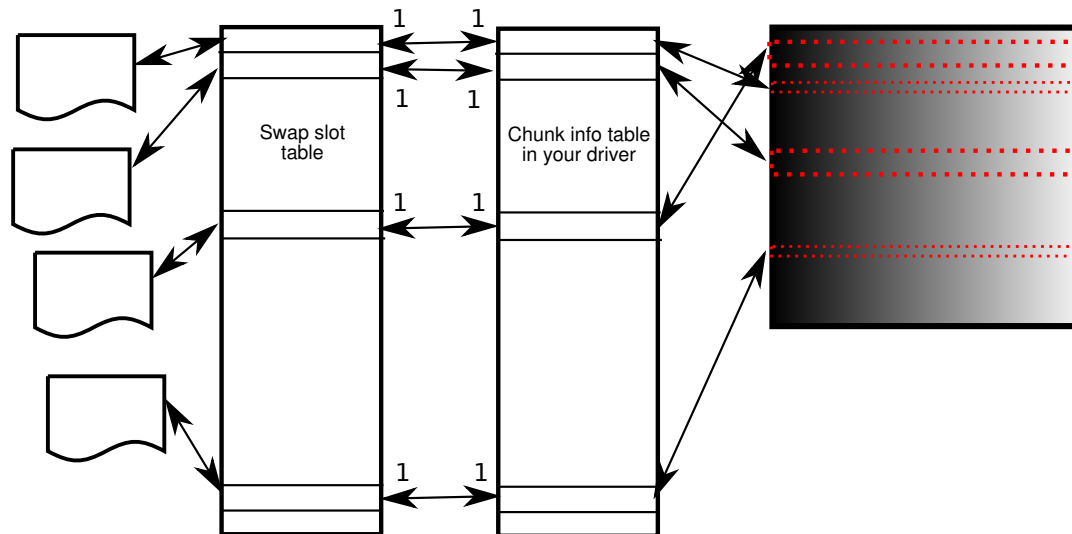
Normal swap device



Your swap device now



Your swap device at the end



Suggested interface

➔ Initialise:

```
int srd_minit(unsigned int totalsize, void *start)
```

➔ Allocate:

```
void *srd_malloc(unsigned int num_bytes)
```

➔ Free:

```
void srd_mfree(void *mem, unsigned int num_bytes)
```

➔ Cleanup:

```
void srd_mcleanup(void)
```

You are expected to write and test this in usermode!

Allocator requirements

- ➔ Keep track of blocks of free memory
- ➔ Ideally avoid splitting up a large free block if there is a smaller free block which satisfies request
- ➔ Balance against performance — e.g. defragmenting every time is too expensive
- ➔ Current scheme is very fast but useless

List ordering

- ➔ **Memory location (address)** — Can coalesce easily
- ➔ **Increasing size** — best [tightest] fit
- ➔ **Decreasing size** — ‘worst fit’, allows fast allocation but with high fragmentation
- ➔ **Increasing time since last use** — Quick to free, but other properties pretty poor in general

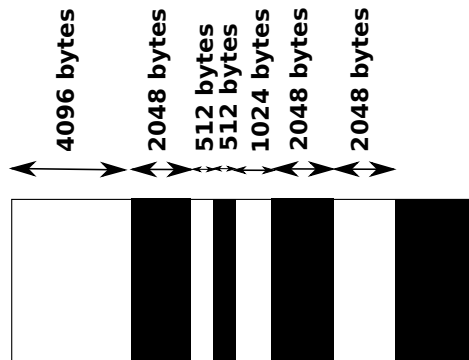
Option two, perhaps with a periodic coalescing phase, is a good bet

Buddy algorithm

- ➔ Effectively is many *address-ordered* free lists, each for *different sized* blocks
- ➔ Combines best of first two options on last slide
- ➔ Relatively high performance coupled with low fragmentation — is a good compromise
- ➔ More complicated to code than a simple free list though
- ➔ Linux kernel uses a (much more complicated) version of this algorithm

Bitmaps

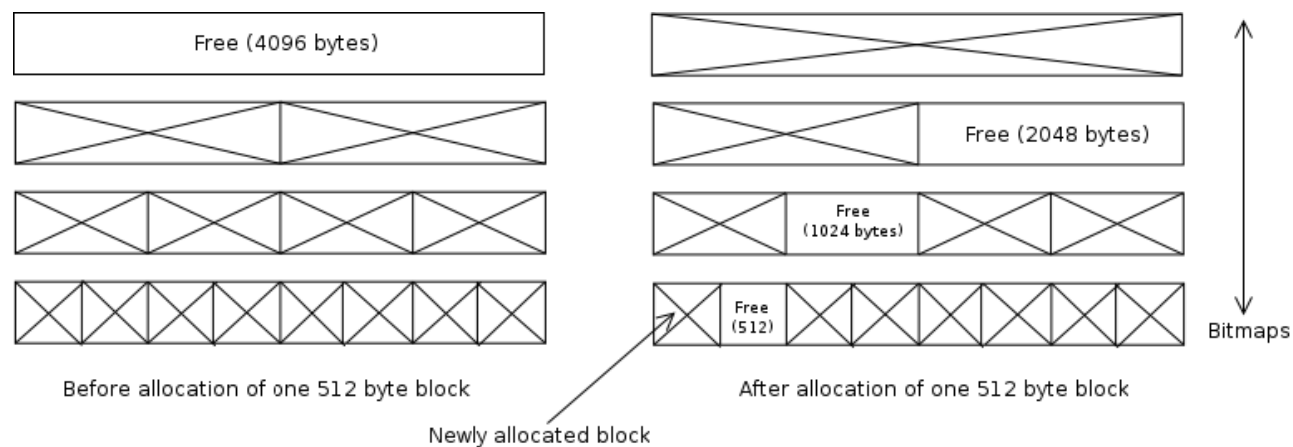
- ➔ A number of address-ordered bitmaps, each for different block sizes, increasing by a factor of two, e.g. 512, 1024, 2048, 4096 bytes
- ➔ **Unset** bit \Rightarrow corresponding block free **and** not contained in larger free block
- ➔ **Set** bit \Rightarrow corresponding block occupied **or** part of larger free block



```
4096:      0      1      1
2048:    1  1  1  1  1  1  1  0  1  1  1  1
1024:   1  1  1  1  1  1  1  0  1  1  1  1
512:    11111111111111011111111111
```

Allocation

- ➔ Compute smallest possible block size
- ➔ Search that bitmap, use if possible
- ➔ Otherwise, try next largest bitmap and repeat until we find a free one
- ➔ Split it into two blocks of the next size down, propagate down



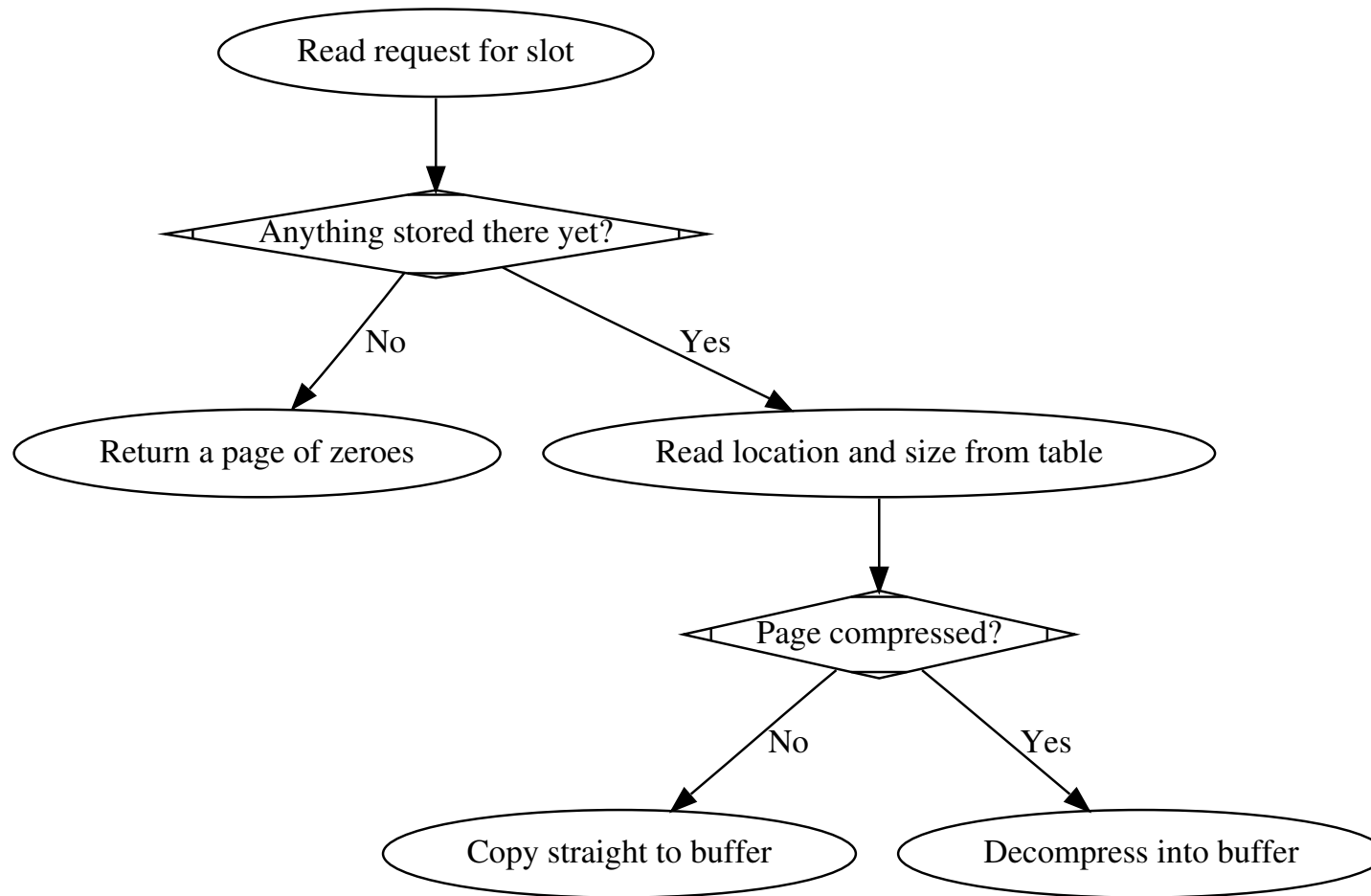
Freeing

- ➔ When a block is freed, check if the freed block's 'buddy' is also free
- ➔ If so, coalesce them
- ➔ Repeat this on the new free block of the next size up, coalescing as far up as possible

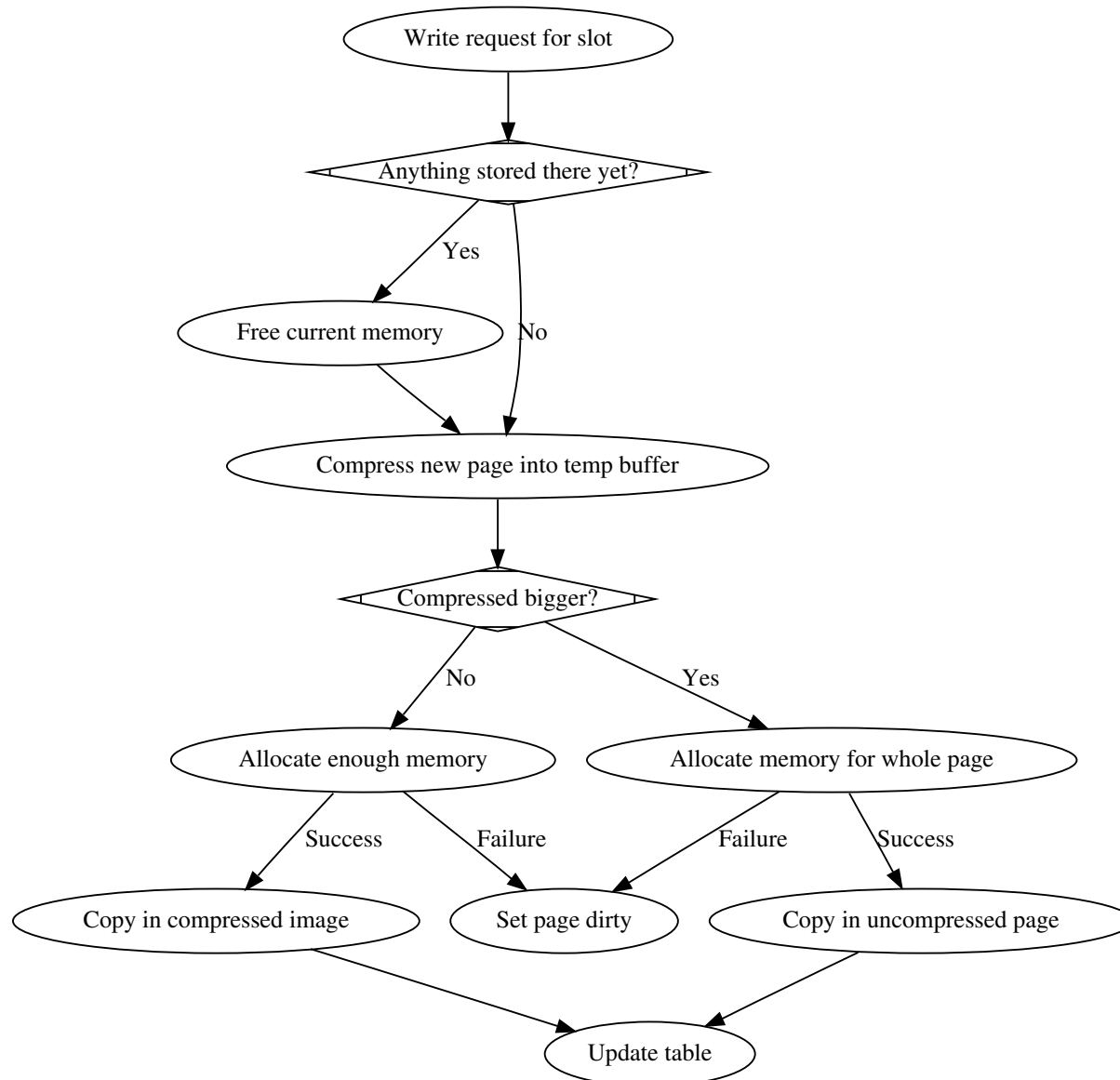
Technicalities

- ➔ If you need to allocate memory for housekeeping structures, use `kmalloc()` for allocations less than 131,072 bytes, `vmalloc()` otherwise (shouldn't be necessary)
- ➔ `ffz()` arch-dependent function useful for efficient bitmap searching

Interfacing the allocator — reads



Interfacing the allocator — writes



Page slot free handler

Need to set this up now

- Called by the kernel when a page slot is freed from the swap device (e.g. all processes referencing it are killed)
- Check the appropriate table entry for that slot and free any memory still allocated to it

Race conditions

Page slot free handler and request queue processing tasklet can race in two cases:

- ➔ Simultaneous execution on a multi-processor machine
- ➔ Page slot free handler can be interrupted by an IRQ and tasklets are often run after processing an interrupt

Protect memory allocator's data structures with a spinlock.

The `spin_(un)lock_bh()` functions (en/dis)able deferrable functions (incl. tasklets, also called **bottom halves**)

Adding some more stats (1)

Finally, we add a few more `ioctl()` handlers so that usermode processes can get stats on your driver...

- ➔ `SRDGETMEM` — request total amount of actual allocated memory (bytes)
- ➔ `SRDGETUSEDMEM` — total amount of actual memory in use (bytes)
- ➔ `SRDGETSLOTS` — total number of page slots your ramdisk advertises, virtual size
- ➔ `SRDGETCOMMITTSLOTS` — total number of page slots currently storing pages (i.e. have memory allocated to them)

Adding some more stats (2)

- ➔ Specific to your device, add constants to a file *include/linux/srd.h*, see the spec for which values to use
- ➔ Don't compute these values each time, maintain rolling counters
- ➔ Use the `atomic_t` atomic integer type
- ➔ Support atomic update via `atomic_read()`, `atomic_add()` etc. in *include/asm-i386/atomic.h*

Assessment/testing

- ➔ Test utility is `swapcheck` again, execute `swapoff; mkswap srd; swapon srd` beforehand

Swapcheck modes

swapcheck has two modes:

- ➔ *Assessment mode*: two parameters; device node and 'y' or 'n', 'y' for milestone 5. Runs for four minutes, processing large amounts of data, fails if detects any corruption or crashes in any way, otherwise, passes
- ➔ *Manual mode*: two parameters; amount of memory to use and 'y' or 'n', 'y' for milestone 5. Runs indefinitely, processing specified amount of data, fails if detects any corruption or crashes in any way, otherwise, passes

The 'y' and 'n' simply let `swapcheck` know if you have a memory allocator yet, so how 'hard' to push your device