BSYS Challenge: Kapitel 21 & 22 (01.12.2020)

Aufgabe 1)

Erklären Sie das obige Diagramm:

Was ist hier dargestellt?

Das Verhalten des Swap Daemon (`kswapd`) bei immer weniger freien Pages in einem System bei Erreichen bestimmter Watermarks.

Wie funktioniert das dargestellte System im Detail?

Es stehen drei Bereiche zur Verfügung: `high_pages` (High Watermark), `low_pages` (Low Watermark), `min_pages`. Diese dienen dem Swap Deamon dazu beim Überschreiten eines bestimmten Werten zu reagieren und Pages auszulagern.

Während des Systemstarts wird ein Kernel-Thread namens `kswapd` aus `kswapd_init()` gestartet, der kontinuierlich die Funktion `kswapd()` in `mm/vmscan.c` ausführt, die normalerweise schläft. Dieser Swap Deamon ist für das Reclaiming von Seiten verantwortlich, wenn der Speicher knapp wird. Ursprünglich hat `kswapd` alle 10 Sekunden darauf gewartet, heutzutage wird er vom physischen Page-Allocator nur dann aufgeweckt, wenn die `low_pages` Anzahl freier Pages erreicht ist.[1]

Wie man auf dem Schaubild erkennen kann, können Pages zunächst normal allokiert werden, wobei die Anzahl der freien Pages auf dem Speicher stark abnimmt.

Sobald `low_pages` erreicht wird und dementsprechend nur noch eine bestimmte Anzahl an freien Pages zur Verfügung steht, wird der Swap Deamon vom Buddy Allocator geweckt, um Pages zu swappen. Diese werden im Hintergrund auf die Disk geswapped, während weitere neue Pages allokiert werden. Das führt dazu, dass die Rate der verwendeten Pages verlangsamt wird, allerdings werden immer noch zu viele Pages allokiert, als geswapped werden können, da der Swap Deamon nur eine bestimmte Anzahl an Pages gleichzeitig auf die Disk swappen kann. [2]

Wenn `min_pages` erreicht wird, übernimmt der Kernel-Allocator die Funktion des Swap Deamons. [3] Dieser kann auch unter der `min_pages` Markierung allokieren und befreit in diesem Bereich synchron.

Der Swap Deamon swapped weiterhin Pages und macht dies solange, bis die Markierung high_pages erreicht worden ist und dementsprechend viele Pages wieder frei sind und als „balanciert“ betrachtet werden können. Danach wird der Swap Deamon wieder schlafen gelegt.

[1] https://www.kernel.org/doc/gorman/html/understand/understand013.html
[2] http://www.science.unitn.it/~fiorella/guidelinux/tlk/node39.html#SECTION00623100000000000000
[3] https://www.kernel.org/doc/gorman/html/understand/understand005.html

Wie Verhalten sich die Funktionen `malloc()` und `free()`, wenn sich das System im grünen | gelben | roten Bereich befindet?

- **Grün und gelber Bereich:** Bevor min_pages erreicht wird werden mit malloc() Pages allokiert. Nach min_pages werden Pages mit free() wieder freigegeben.
- **Roter Bereich:** malloc () und free () laufen synchron: Unterhalb der min_pages Markierung wird malloc() blockiert bis eine entsprechender Bereich von free() wieder freigegeben wird., manchmal auch als `direct-reclaim` bezeichnet.
  Wenn eine memory allocation request nicht aus der free list erfüllt werden kann, versucht der Kernel Pages direkt im process context, der die memory-allocation vornimmt, zurückzufordern. Das Umlenken einer allocation request in diese Art von Clean Up wird als " direct reclaim" bezeichnet. [4]

Bewerten Sie diese Art der 'Background' Work. - Kennen Sie weitere Beispiele von Background Work in einem Betriebssystem (Beispiel+Erklärung)?

Swaps können gesammelt und gemeinsam auf die Disk geschrieben werden. Durch Clustern bzw. Gruppieren von Pages, die geswapped werden sollen, werden die Seek bzw. Rotations-Eigenschaften einer Disk effizienter genutzt.

- `init:` Unix-Programm, das alle anderen Prozesse startet. Ab 2016 wurde es für die wichtigsten Linux-Distributionen durch systemd ersetzt.

- `crond:` Zeitbasierter Job-Scheduler, führt Jobs im Hintergrund aus.

Aufgabe 2)

Aging:

Ähnliches Verhalten wie LRU, aber bei jedem Pagezugriff wird die Zeit gemessen, die der Zugriff gedauert hat. Eine Liste wählt das mit dem lowest Counter aus.

Verhält sich gut bei 80-20 Workloads, da es die Möglichkeit gibt, aus dem vorherigem Verhalten zu lernen. Bei Looping kommt es wie bei LRU auf die Größe des Caches an. Allerdings mit dem Unterschied zwischen Aging und LRU, dass Aging die Referenzen nur in den letzten 16/32 Zeitintervallen verfolgen kann. Im Allgemeinen reicht es aus die Inanspruchnahme der letzten 16 Zeitintervalle zu kennen, um eine gute Entscheidung treffen zu können, welche Pages geswapped werden sollen. [5]

Longest Distance First (LDF):

LDF ist ähnlich wie LRU, nur dass der Unterschied darin besteht, dass in LDF die Locality auf der Entfernung und nicht auf den verwendeten Referenzen basiert. Es wird die Page ersetzt, die am weitesten Entfernt ist von der aktuellen Page.

LDF ist gut für einen 80-20 Workload, da dieser Workload Locality aufweist.

---

[4] https://lwn.net/Articles/396561/
[5] https://en.wikipedia.org/wiki/Page_replacement_algorithm

Bei einem Looping Workload kommt es auf die Cache Size, so wie bei LRU, an.

Bei LRU wird bei einem Looping Workload eine Hitrate von 0 % erreicht, solange die Cache Size kleiner ist als die zu referenzierenden Pages und stellt damit ein worst-case Szenario dar.

Das worst-case Szenario beim Looping-Workload bei LRU entsteht dadurch, dass die am wenigsten verwendeten Pages bereits früher aufgerufen werden als die Pages, die LRU lieber im Cache halten möchte.

Da LDF auch nach Locality entscheidet, welche Page zu ersetzten ist, verhält es sich hier wie bei LRU.

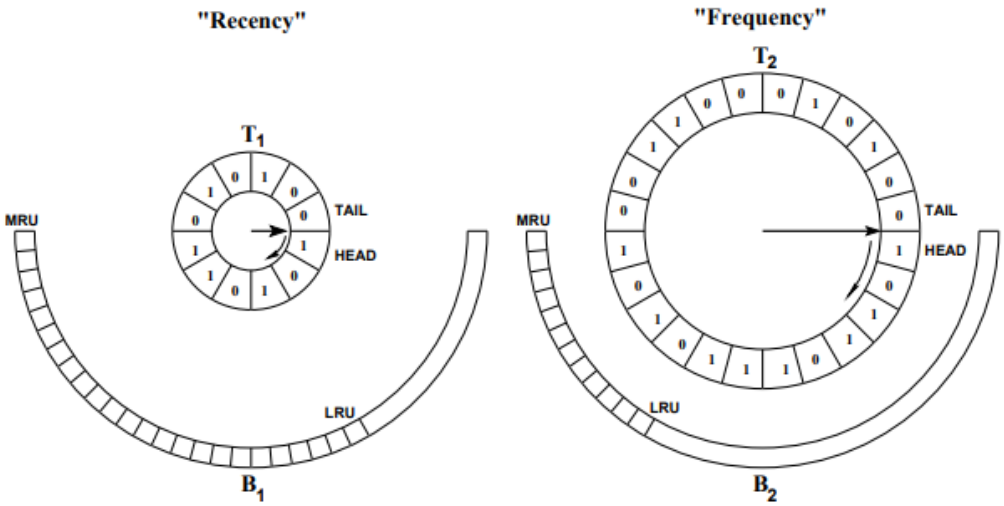
Clock with Adaptive Replacement (CAR):

Basiert auf CLOCK, erfasst aber dynamisch die Aktualität und Häufigkeit eines Workloads.

Performt besser als LRU und CLOCK bei einem 80-20 Workload, da die Häufigkeit mit in Betracht gezogen wird und braucht keine User Parameter.

Bei einem Looping-Workload verhält es sich bei CAR wie bei CLOCK schlecht, da die Looping Eigenschaft bei kleinerer Cache Size als Anzahl referenzierender Pages dazu führt, dass die die Use- bzw. Reference-Bits beeinflusst werden und das Erfassen der Aktualität und Häufigkeit nicht gewährleistet werden kann.

[6]Beispiel zu CAR:



Die CLOCKS T1 und T2 enthalten die Pages, die sich im Cache befinden, sowie die Listen B1 und B2, welche historische Pages enthalten, welche kürzlich aus dem Cache ausgelagert wurden. CLOCK T1 erfasst die Aktualität „Recency“ und CLOCK T2 die Häufigkeit „Frequency“. Die Listen B1 und B2 sind LRU-Listen. Pages, die aus T1 ausgelagert wurden, werden auf B1 gesetzt und die Pages, die aus T2 ausgelagert wurden, werden auf B2 gesetzt. Der

[6] https://www.usenix.org/legacy/publications/library/proceedings/fast04/tech/full_papers/bansal/bansal.pdf

Algorithmus versucht B1 in etwa so groß wie T2 und B2 in etwa so groß wie T1 zu halten. Der Algorithmus begrenzt auch das Überschreiten der Cache Size durch |T1| + |B1|. Die Größen der CLOCKS T1 und T2 sind an eine wechselnde Workloads angepasst. Wenn es einen Hit in B1 gibt, wird die Zielgröße in T1 um eins erhöht und T1 verringert, wenn es einen Hit in B2 gibt. Neue Pages werden entweder auf T1 und T2 direkt hinter den Zeiger gesetzt, welcher sich im Uhrzeigersinn dreht. Bei neuen Pages wird das Reference-Bit auf 0 gesetzt. Bei einem Cache Hit zu einer beliebigen Page wird in T1 und T2, je nach dem, wo sich die Page befindet, das Reference-Bit auf 1 gesetzt. Wenn der T1 Zeiger auf eine Page mit Reference-Bit gleich 1 trifft, wird die Page hinter den T2 Zeiger gesetzt und das Reference-Bit auf 0 gesetzt. Wenn der T1 Zeiger auf eine Page mit Reference-Bit gleich 0 trifft, wird die Page ausgelagert und an der MRU-Position in B1 untergebracht. Wenn der T2 Zeiger auf eine Page mit Reference-Bit gleich 1 trifft, wird das Reference-Bit auf 0 zurückgesetzt. Wenn der T2 Zeiger auf eine Page mit Reference-Bit gleich 0 trifft, wird die Page ausgelagert und an der MRU-Position in B2 untergebracht.