2013 European Control Conference (ECC)
July 17-19, 2013, Zürich, Switzerland.

Feedback-based memory management with active swap-in

Federico Terraneo[1] and Alberto Leva[2]

Abstract— **Memory and swap management is an important issue in operating systems, owing to the large difference between RAM and disk access times. This paper presents a memory management layer, designed along a fully control-theoretical approach, that decouples the swap-out and swap-in activity from the events triggered by the applications' memory access patterns. A system endowed with this layer can manage memory on a per-process basis, while at the same time avoiding the presence of swapped-out pages if RAM memory is available. Such a tunable active swap-in mechanism thus inherently avoids temporary RAM over-utilisations to slow down the system for a period significantly longer than their duration. Simulation results prove the effectiveness of the proposal, as well as the viability of its integration with existing memory managers.**

I. Introduction

Modern operating systems manage memory through a paging mechanism, and encompass a swap space to allow the pool of running processes to collectively access more memory than the available RAM.

Traditionally, the action of swapping memory pages out to disk, or *vice versa* from disk into RAM, is dictated solely by the memory access patterns of the processes. Therefore, it may happen that a memory area remains swapped to disk for a long time even if RAM space is available. This has basically two unpleasant consequences. First, the fact itself just mentioned can slow down the system. Second and most important, the swap-in of that space into RAM may occur when the CPU is highly loaded, while from the swap-out instant till the swap-in request there may have been plenty of time with a low load. Apparently, if an active swap-in mechanism were available to free the swap space when there is room in RAM and the CPU is not too loaded, a benefit for the overall system responsiveness would be gained.

Another relevant problem is that memory occupation is a system-wide fact, while responsiveness is best viewed at the application level. For example, if a memory-intensive noninteractive application is run in parallel with a window manager, the memory needs of the former may well cause pages from both to swap to disk, despite the latter suffers of the resulting slowdown to far more significant an extent. In this respect, a memory manager should take its decisions, including the swapping-related ones, on a per-process basis.

As the following literature review will point out, the initial research on virtual memory, carried out in the early days of multitasking computers, has since faded and current operating systems are essentially uniform in the way they implement virtual memory. It is nowadays taken for granted that swapping in and out needs to be triggered by application events only, such as memory requests and page faults. In this case, the only decision to be taken is *which* pages to swap-out when RAM is not sufficient, with the Least Recently Used (LRU) policy being the most common choice. In the opinion of the authors, there are two more fundamental question to be answered, which are *who* and *when*. The first question is related to per-process memory (limit) set points, and could be used to achieve *memory access temporal isolation*, while the second question opens the doors to transfers between swap and RAM tha are time-triggered instead of event-triggered by process page faults.

To the best of the authors' knowledge, no integrated solution for the issues above is at present available. This work shows how one can be devised with a completely control-theoretical approach, i.e., as a feedback controller that allow for a formal assessment of the obtained control system as for performance and robustness.

II. Related work and motivation

The introduction of virtual memory dates back to the first multitasking computers. A major advantage of this technology is to allow programs to allocate more memory than the amount of RAM available in the system.

Owing to the large difference between the RAM and disk access times, different replacement strategies were explored for swapping process memory to disk. One of the first works on the subject, namely [3], proposed an on-demand page swapping, that is called when a page fault occurs to swap the required page into memory. In this occasion, and when memory allocation occurs, if the physical memory is exhausted, one or more pages are swapped out. Different algorithms are proposed to select which page(s) to swap-out, including a round-robin scheme. Despite the publication dates back to 1969, this scheme is surprisingly similar to what is used nowadays, with the exception of missing the LRU policy.

In [1] a different solution is presented, which consists in swapping out entire applications (and suspending them) in case the RAM memory is not enough. This solution does not perform swap-in based on page faults, as a suspended application cannot produce any, but decides to reload an application when the available memory becomes sufficient again. This solution is evidently targeted at optimising throughput for batch jobs.

Another approach is considered in [4]. This solution can both swap in and out entire applications for batch jobs, or

[1]F. Terraneo is a PhD student at the Politecnico di Milano, Dipartimento di Elettronica e Informazione terraneo@elet.polimi.it

[2]A. Leva (corresponding author) is with the Politecnico di Milano, Dipartimento di Elettronica e Informazione, Milano, Italy leva@elet.polimi.it

single pages for interactive ones. The work introduces a notable concept, the *resident set* which can be thought of as an hard limit on the amount of memory a process can allocate in RAM. If this limit is reached every new allocation will cause the swap-out of pages taken from the *same* process. This solution explicitly addresses the problem of memory access temporal isolation, as an application making an excessive use of memory cannot swap-out pages from other processes, causing unpredictable system-wide slowdowns.

After that first research effort, however, the swapping behaviour of virtual memory systems was consolidated to a demand paging with LRU-based page replacement, and the literature on this subject became silent, except some sporadic publications such as [8]. In that paper the pagefault-driven swapping-in is augmented by relaxing the constraint that only the page that caused the fault needs to be loaded in memory. A prediction of pages likely to be referenced based on the faulted one is used to load additional pages.

Other more recent research directions regard the introduction of garbage collection and other wear-levelling schemes to allow implementing a swap over a FLASH memory instead of a disk, which has the issue of limited number of write cycles [2], [5], and on distributed virtual memory for use in data centres [6], [7]. To the best of the authors' knowledge, all recent publications did not question the foundation of virtual memory, and hardly any further research effort was spent on this crucial point of computing systems. This is also reflected by contemporary operating systems, which employ the same on-demand paging system based on the LRU scheme introduced long ago.

In the opinion of the authors there are, however, many significant use cases where this consolidated scheme fails to provide optimal performances. The first issue is related to the system-wide nature of the LRU scheme currently adopted. A typical use case is when a memory-intensive background task is run concurrently to some interactive task, such as a backup application and a windowing environment. As the background task allocations cause the exhaustion of the available RAM, the LRU schemes will swap-out pages from arbitrary running processes, including the interactive ones, causing a noticeable reduction in the system responsiveness. This is caused by the lack of a per-process management policy that can control *who* are the processes that have exceeded their memory limit and have to be selected as targets for swap-outs.

Another use case that points out the issues of a purely demand paging system is when a process transiently allocates a large amount of memory. In this case part of that process will be swapped and, due to the system-wide LRU scheme, also part of other processes will. When the complex task ends, the memory occupation drops sharply, resulting in a large amount of free RAM. If in this situation the system is left idle for an arbitrary amount of time, it will not completely recover due to swap-in being only triggered by application page faults. This clearly exacerbates the lack of control about *when* to perform swap-in.

This works proposes a perspective change, that explicitly focuses on the two aforementioned problems related to virtual memory, *who* and *when*. The result is a controller that extends a traditional kernel memory manager introducing two previously absent functionalities: handling per-process memory limits and automatic swap-in. As introduced here, this solution bears some resemblance with the VAX/VMS one, but a significant difference is that the hard-limit of the VAX approach is here substituted with a soft limit. In the proposed scheme a process can exceed its memory limit so long as there is enough memory in the system. As memory is always partitioned in a way whose sum amounts to the available RAM, for this to happen it means that other applications are using less memory than their limit. If the applications below their limit then allocate more memory and as a result the available one is exceeded, this causes the applications above their limit to swap. Therefore, unlike in the VAX solution it is not true that memory allocations cause swap-out only from the same process. On the contrary, memory allocations cause swap-out only from processes above their limit. In addition, the proposed solution performes automatic swap-in and is based on a control-theoretical approach. These features further set this solution apart from the literature.

III. Control problem statement

At the level of depth required for this treatise, the "traditional" memory management can be represented as in figure 1 with the exception of the part indicated as "new elements". A set of processes allocate and deallocate memory. This can be done explicitly, through the `brk()` and `mmap()` system calls, which are used e.g. by the `malloc()` and `free()` C library functions. It can also occur implicitly, when a stack growth causes the need to allocate to the process one or more memory pages (this is called a *page fault*). Memory *requests* are served by the *kernel allocator*, an operating system component that has also the task of moving memory pages from RAM to swap space in case no available RAM is found. Memory *access* by processes is conversely served by a Memory Management Unit (MMU), a hardware component that performs the translation between virtual and physical addresses. If a memory access falls in a page that has been swapped out, the MMU causes a page fault and the kernel allocator is called to load that page into RAM.

A. Controlled system model

From the point of view of allocated RAM and swap, the generic (i-th) process can be represented by the discrete-time, linear and time invariant model

$$\begin{cases} m_i(k+1) &= m_i(k) + a_i(k) - dm_i(k) + pf_i(k) + u_i(k) \\ s_i(k+1) &= s_i(k) - ds_i(k) - pf_i(k) - u_i(k). \end{cases} \tag{1}$$

where $m_i(k)$ is the amount of allocated RAM memory and $s_i(k)$ that of allocated swap. The index k counts each memory-affecting operation, thereby making (1) discrete-time but not sampled-signals, and the other terms describe all the possible causes for a memory state change. In detail, in a

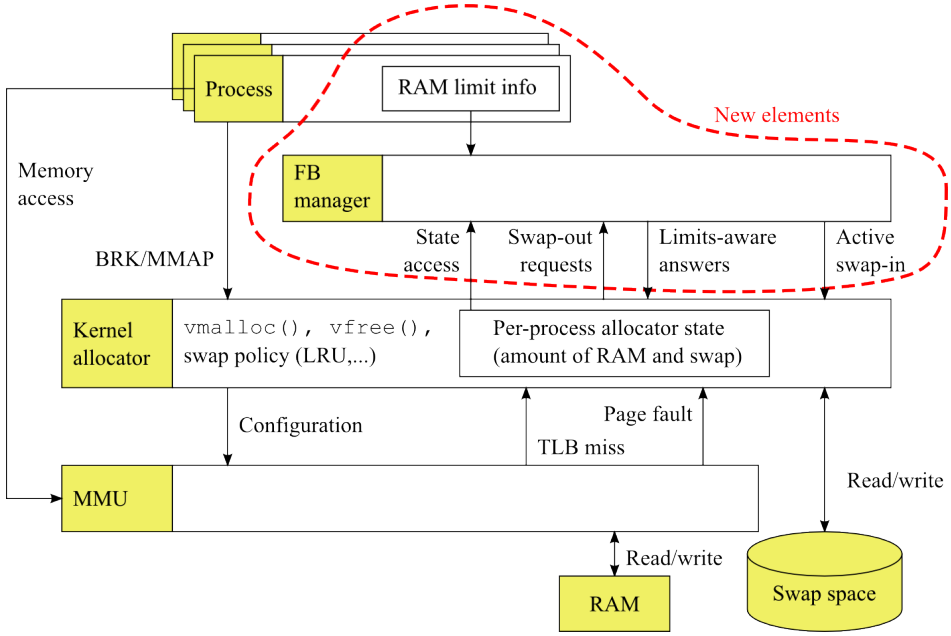

Fig. 1. Traditional memory management, and the proposed improvement.

certain memory operation, a process can allocate a quantity $a_i(k)$ of memory, and since it is reasonable that it will be used very shortly, the kernel allocator will make sure to allocate it in RAM.

A process can also deallocate memory, corresponding to $dm_i(k)$ if this is RAM, or $ds_i(k)$ if it is swap (the allocator can obviously free swap memory without intermediately moving it to RAM). The term $pf_i(k)$ represent the page faults that a process can generate in a highly unpredictable manner, depending on its memory use pattern, and acts symmetrically on RAM and swap. Note that all the quantities mentioned so far are physically bound to be nonnegative. Note also that all are known by the kernel allocator, i.e., they are measurable without error.

Finally, $u_i(k)$ represents the action of moving memory from RAM to swap or *vice versa*, that is performed by the allocator; $u_i(k)$ can therefore take both positive and negative values. The resulting model is therefore composed of two discrete integrators per process, subject to the physical constraints

$$\begin{cases} m_i(k) & >= & 0 \\ \sum_i m_i(k) & < & \beta\overline{M} \\ s_i(k) & >= & 0 \\ \sum_i s_i(k) & < & \overline{S}, \end{cases} \tag{2}$$

where $\overline{M}$ and $\overline{S}$ are the maximum amount of memory and swap in the system, while β takes into account that some of the physical memory may be reserved, for example by the operating system itself. The $\beta\overline{M}$ term is here called *global maximum memory occupation*. For completeness, if both memory and swap are exhausted, there is another component of the operating system – named the *out of memory killer* – that terminates processes to free up memory. Such an event is however considered a pathological system condition, indicating a malfunction of some process – that should occur very sporadically – or an erratic swap space configuration on the part of the system administrator. Both cases are apparently not to be dealt with by the memory manager, thus not addressed herein: for our purposes, in other words, the swap space can be considered infinite.

The physical constraints (2) are partly naturally enforced by the system. For example, if a process has no swap, it cannot generate page faults, and it cannot deallocate memory it does not have (if programming errors causes a program to attempt that, the kernel detects the error and terminates the process before the memory system is set to an inconsistent state).

Memory allocations are conversely unconstrained, hence the system cannot work in the total absence of control, here represented by $u_i(k)$, that is constrained by

$$\begin{cases} u_i(k) & >= & -m_i(k) - a_i(k) + dm_i(k) - pf_i(k) \\ u_i(k) & <= & s_i(k) - ds_i(k) - pf_i(k), \\ \sum_i u_i(k) & <= & \beta\overline{M} \\ & & -\sum_i \left(m_i(k) + a_i(k) - dm_i(k) + pf_i(k)\right) \end{cases} \tag{3}$$

Finally, the kernel handles memory in terms of pages, which are a set of contiguous memory locations, with a typical size being 4KBytes. Therefore, all quantities in the model are expressed in memory pages, and are meaningful only if integers.

B. Requirements

The currently available memory managers are not designed as controllers. They simply act in response to process-originated requests, having the sole aim of respecting (3). However, *everything* is done at the system level. In particular, the decision on *which* process will have its memory swapped

out (i.e., the value of the $u_i(k)$ once the constraint on their sum is obeyed) is taken based on policies like the Least Recently Used (LRU) one.

Interestingly enough, the proposed model can represent such managers. It suffices to assume $m_i(k)$, $a_i(k)$, $dm_i(k)$, $ds_i(k)$ and $pf_i(k)$ as disturbances – as they *de facto* are from the memory manager's standpoint – and add some logic to represent e.g. the LRU policy. However, it has already been shown that such managers perform sub-optimally in some relevant cases. The idea is here to operate for the $u_i(k)$ entirely on a per-process basis, and to introduce an additional input to cause the system to proactively free the swap space when possible.

As it will now be shown, this can be done with a very simple control structure, ultimately determining *how much* memory to swap in or out for each process. Within the process one can clearly continue using LRU-like policies to decide *what* to swap, hence the added control layer is transparent in that respect.

To design the mentioned layer, the following requirements, in addition to (3), need fulfilling:

- if there is not enough RAM for all the processes, pages will be swapped out only from those that are exceeding a given limit (that can be different for each process);
- if there is available RAM and occupied swap, the latter has to be reloaded in RAM at a system-wide specified rate, unless swapping in is explicitly required by some process.

IV. Control synthesis

The requirements just expressed can be summarised by saying that when there is RAM available, the system has to behave like an open-loop (vector) integrator, divergence (and in particular the violation of the $m_i(k), s_i(k) >= 0$ constraint) being avoided by the assumed sanity of the processes behaviour, and in addition active swap-in has to occur if possible. Notice that unless page faults are occurring – i.e., if no process is complaining about memory – there is no need in this situation to enforce the prescribed RAM distribution. On the contrary, when the required RAM is not available, swap-out has to occur in such a way to recover the required RAM distribution, i.e., not penalising "important" processes.

In figure 2, the first component is the prediction of the memory status at the next $(k+1)$ step, consisting of $M(k)$ plus the sum of RAM disturbances $\delta m(k)$, which collects $a(k) - dm(k) + pf(k)$, plus an input vector u_s that serves to dictate the active swap-in rate. The sum of these values is the total memory occupation at the next step, in the absence of control. That value is compared with the (limit) set point $\beta\overline{M}$ and passed through a saturation at zero.

This results in a control loop that is closed only if the memory state at the next step would exceed the limit. In this case, the saturation passes this negative value, which amounts to the RAM that needs to be swapped out, to a nonlinear function f whose purpose is to distribute that amount of memory among processes, considering per-process limits. A possible selection for this nonlinear function will be shown later, but for now it is important to note that, for the control scheme to work, the sum of the (integer) elements of its output vector $\widetilde{u}$ should be equal to its input $\widetilde{e}$. The active swap-in vector u_s can be normally zero and periodically set to have all positive elements.

To analyse this control scheme, it is possible to focus on the transfer function between $\widetilde{e}$ and M, that makes sense assuming no saturation is active. Taking as state vector

$$\underline{m}(k) = \left[\sum_i m_i(k)\ m_2(k)\ \ldots\ m_n(k)\right]' \tag{4}$$

results in the system

$$\begin{cases} \underline{m}(k+1) &= \underline{m}(k) \\ & \quad + \begin{bmatrix} \sum_i(\delta m_i(k) + u_{si}(k) + f_i(sp_i(k), m(k), \widetilde{e}(k)) \\ \delta m_2(k) + u_{s2}(k) + f_2(sp_2(k), m(k), \widetilde{e}(k)) \\ \vdots \\ \delta m_n(k) + u_{sn}(k) + f_n(sp_n(k), m(k), \widetilde{e}(k)) \end{bmatrix} \\ M(k) &= [1\,0\ldots 0]\ \underline{m}(k). \end{cases} \tag{5}$$

Now, as already stated the nonlinear function f has to be selected so that $\sum_i(f_i(\alpha_i(k), m, \widetilde{e}(k)) = \widetilde{e}(k)$. Considering this, and setting $delta = \sum_i(\delta m_i(k) + u_{si}(k))$, the system can be rewritten as

$$\begin{cases} \underline{m}(k+1) &= \underline{m}(k) \\ & \quad + \begin{bmatrix} \delta + \widetilde{e}(k) \\ \delta m_2(k) + u_{s2}(k) + f_2(sp_2(k), m(k), \widetilde{e}(k)) \\ \vdots \\ \delta m_n(k) + u_{sn}(k) + f_n(sp_n(k), m(k), \widetilde{e}(k)) \end{bmatrix} \\ M(k) &= [10\ldots 0]\ \underline{m}(k). \end{cases} \tag{6}$$

In this model only the first state variable, which represents the sum of the RAM memory allocated by all processes, is observable, and it is not influenced by the nonlinear function, whose only purpose is to partition the swap-out signal among processes. This allows to employ for the purposed of this section the simplified model of figure 3.

This model has the only state variable $M(k)$, and reads

$$M(k+1) = M(k) + d(k) + min(0, \beta\overline{M} - \delta(k) - M(k)) \tag{7}$$

The system thus works in two possible ways, depending on the $min()$ saturation.

$$\begin{cases} M(k+1) &= M(k) + d(k) &: \quad M(k) + d(k) < \beta\overline{M} \\ M(k+1) &= \beta\overline{M} &: \quad M(k) + d(k) >= \beta\overline{M} \end{cases} \tag{8}$$

As can be seen, the controller can guarantee that the global RAM limit is never exceeded so long as no saturations in the process occur. Function f has then to avoid such saturations, and once this is guaranteed, its presence is irrelevant for the control loop. For completeness and to avoid lengthy explanations, a possible implementation of f is here reported as Scilab code

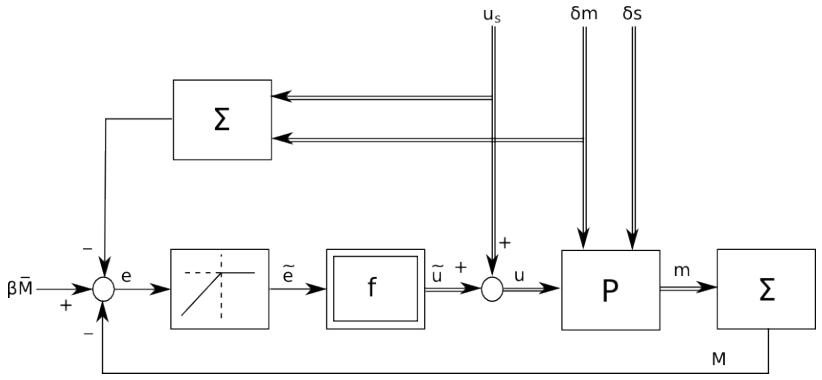

Fig. 2. The proposed control scheme.

```
function result = integerShare(value,share)
    result = [];
    // compute the process swapout share except for
    // the last one, rounding to the nearest integer
    for i=1:length(share)-1
        result = [result,round(value*share(i))];
    end
    // compute the last so as to preserve the total
    result = [result,value-sum(result)];
endfunction

function result = f(sp,m,etilde,deltam)
    result = zeros(sp);
    totalover = sum(max([0,0,0],m-sp));
    // first try to partition swap-out against
    // processes above their limit
    if(totalover>0)
        result = integerShare(min(etilde,totalover),...
                 max([0,0,0],m-sp)/totalover);
    end
    // if not enough swap-out form processes that
    // are allocating memory right now
    if(etilde>totalover) then
        result = result...
                 +integerShare(etilde-totalover,...
                  max([0,0,0],deltam)/...
                  sum(max[0,0,0],deltam));
    end
end
```

Apart from the problem formulation in control-theoretical terms, a major peculiarity of this work is the role of function f. In fact the controller is executed at every change in the process memory allocation – which is an event-triggered phenomenon – and also periodically, to enforce the required swap-in. Function f takes care of avoiding useless swapping activity by rapidly enforcing memory limits only when necessary, i.e., only when page faults evidence that some process is complaining. Note that without the additional time triggering above this would also apply to active swap-in, however, thereby preventing to prescribe a rate for it independently of the processes' activity, and thus diminishing the scheme efficacy.

V. Simulation results and discussion

Figure 4 shows the results of a simulation with the proposed controller. In the system there are initially four processes, P1–P4 in the figure, each with 100 RAM pages allocated and no swap. The processes start allocating memory, and thus P3 and P4 exceed their RAM limit around

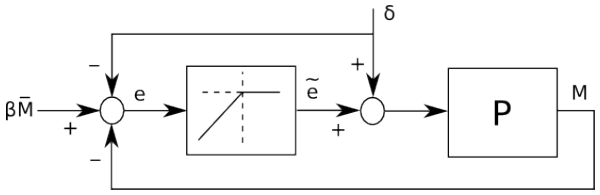

Fig. 3. The reduced control scheme.

$k = 20$ and $k = 50$, respectively. However, since there is still some RAM available, no control action is taken. Then, around $k = 80$, additional memory allocations cause P3 and P4 to start swapping. The other two instead do not swap, and are even allowed to allocate additional RAM, as they is still below their limit: the controller makes room for this in RAM by swapping those above their limit.

Around $k = 200$ the processes stop allocating memory. As can be seen from the bottom row, the RAM is fully utilised.

At $k = 300$ the RAM allocation (limits) set point is changed, but no page faults occur. In this situation the presence of the active swap-in vector causes a slow but sustained change of the processes' RAM towards the new set point. In the presence of page faults, the mentioned change rate would have been faster, thereby accommodating for the applications' desires.

At $k = 600$ the P1 and P4 terminate, causing the immediate deallocation of both their RAM and swap. The availability of additional RAM, coupled with the existence of used swap, triggers the active swap-in vector activity, which causes a swap-in of the two remaining processes, again in the absence of page faults. Around $k = 740$ all of the swap has been brought back into RAM, so that when the processes will access their memory pages and no page faults will occur, significantly increasing their responsiveness.

VI. Implementation related considerations

The modifications that a standard kernel allocator needs to implement the proposed scheme are outlined in figure 1 as "new components". From a software engineering point of view, these are just new functions to access the allocator

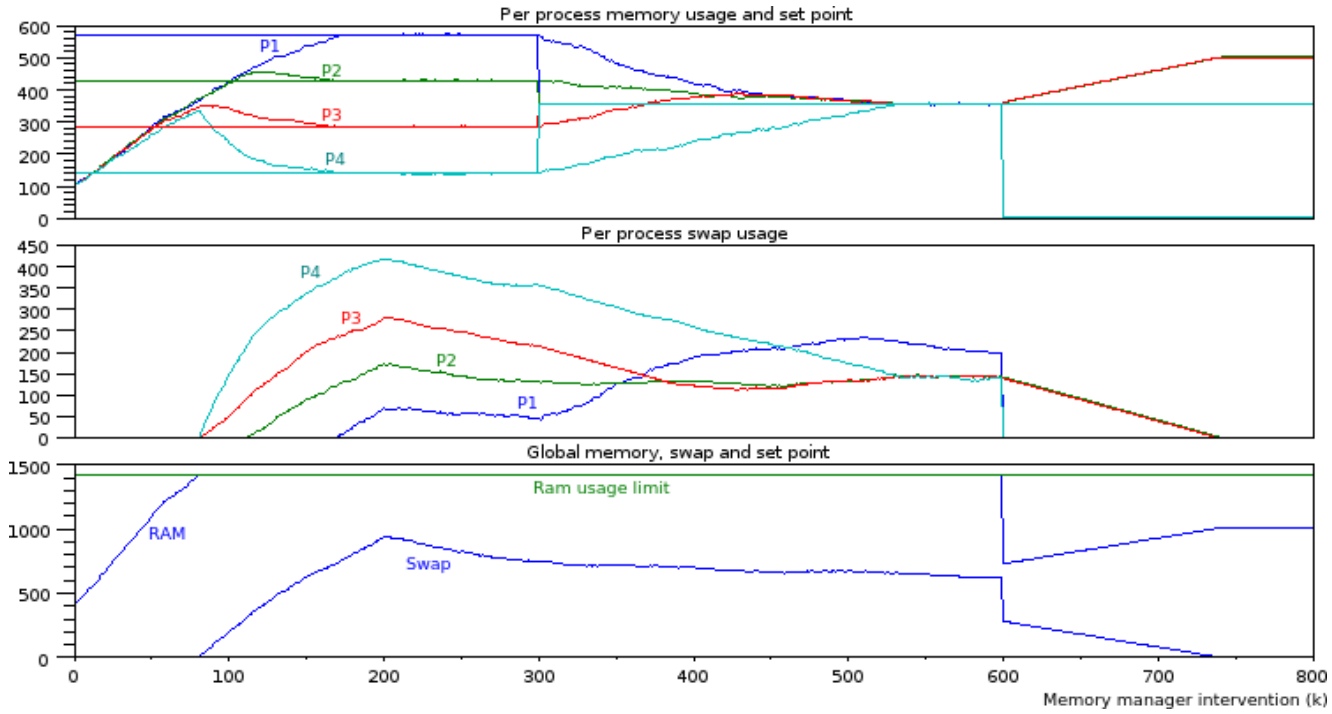

Fig. 4. Simulation results. RAM memory occupation of each process, and limit (top), per-process swap usage (middle), total RAM, swap and limit (bottom)

internal state, notably the amount of RAM and swap of each process.

The feedback manager needs to be called every time memory allocations, deallocations or page faults occur. In particular, when swap-outs are needed, the existing LRU policy needs to be reorganised by letting the feedback manager decide how many pages to swap out from each process; also, the kernel allocator needs a process-wise LRU policy to decide which pages of each process to swap out.

As active swap-in, the feedback manager needs to be periodically called – at a convenient rate – also in the absence of process-triggered memory operations. Note that when a page fault occurs, it happens on a specific swapped page, so the kernel allocator has no degree of freedom regarding which page to swap in. Also, Recently Swapped (LRS) policy is required to choose – at the system level – the pages to swap in, but this is substantially analogous to the existing LRU mechanism.

VII. Conclusions

A memory management system was presented, that can manage memory on a per-process basis, while at the same time avoiding the presence of swapped-out pages if RAM memory is available. Such novel characteristics, with respect to present operating systems, were obtained by formulating the problem as a discrete-time feedback control one. The proposed scheme avoids temporary RAM over-utilisations to slow down the system for a period significantly longer than their duration, and also provides temporal memory access isolation, e.g. by preventing a process with excessive memory requirements to cause a persistent system-wide swap-out.

The scheme is very simple, implementation issues being basically related to the integration into existing kernels. These issues pose hardly any conceptual problem but are quite long to deal with from a technological standpoint. Addressing them, in a view to a full realisation of the scheme, will be the subject of future work.

References

[1] W.M. Chow and W.W. Chiu. An analysis of swapping policies in virtual storage systems. *IEEE Transactions on Software Engineering*, 3(2):150–156, 1977.

[2] S. Ji and D. Shin. An efficient garbage collection for flash memory-based virtual memory systems. *IEEE Transactions on Consumer Electronics*, 56(4):2355–2363, 2010.

[3] R.M. Jones. Factors affecting the efficiency of a virtual memory. *IEEE Transactions on Computers*, 18(11):1004–1008, 1969.

[4] H.M. Levy and P.H. Lipman. Virtual memory management in the vax/vms operating system. *Computer*, 18(3):35–41, 1982.

[5] H.L. Li, C.L. Yang, and H.W. Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):952–964, 2008.

[6] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato. DLM: A distributed large memory system using remote memory swapping over cluster nodes. In *Proc. IEEE International Conference on Cluster Computing CLUSTER'08*, pages 268–273, Tsukuba, Japan, 2008.

[7] H. Midorikawa, K. Saito, M. Sato, and T. Boku. Using a cluster as a memory resource: A fast and large virtual memory on MPI. In *Proc. IEEE International Conference on Cluster Computing CLUSTER'09*, pages 1–10, New Orleans, LA, 2009.

[8] E. Mumolo and G. Bernardis. A novel demand prefetching algorithm based on volterra adaptive prediction for virtual memory management systems. In *Proc. 30th Hawaii International Conference on System Sciences*, volume 5, pages 160–167, Wailea, HI, 1997.