Primary memory

Computer memory is not uniform. On this lecture, we will discuss these types of memory which are fundamental from the viewpoint of the modern architecture (x86); they are used while executing *every* program. Besides the processor registers (already well known to us), these include: cache and RAM. Altogether, they're called **primary storage** (or *primary memory*), though both these terms are also used to describe the RAM layer alone.

Of course, there also exist other memory types, e.g. SSD and HDD drives, pendrives, DVD discs etc. These form the so-called **secondary** (or *external*) **memory**, which will be discussed on a later lecture. Unlike the primary memory, it is not available to the processor directly, but via the input/output interfaces. It's also not strictly *necessary* for executing *every* program, though it's an important component of every personal computer, as all kinds of **non-volatile memory** (keeping its content even after losing electric power) belong to the secondary memory.

Types of primary memory

The main two memory properties, from the user viewpoint, are **capacity** and **speed** (understood as the time needed to reaching the data, but also the amount of data transferred in 1 second). Sadly, these two features are hard to combine, which is caused both by economy (faster memory is more expensive, so available in smaller amounts) and technology (e.g. to have a very quickly reachable cache, we need to place it very close to the MMU, and that puts a limit on its size).

With respect to these two properties, the types of primary memory look as follows:

- **Main memory** (typically called simply **RAM**, or *primary memory*) — largest and slowest.
  It defines the area of memory available directly to the processor (as we already described in the lectures "Computer structure" and "Processor and programs"). On this lecture, we will discuss in more detail the process of translating between logical and physical addresses, and in particular, the important technique of *memory virtualization*.
- **Cache** — intermediate capacity and speed.
  Cache is physically placed inside the processor and acts as a buffer for exchanging data between the RAM and the registers. On this lecture, we will describe its purpose, structure and rules of operation.
- **Processor registers** — fastest and smallest; already discussed in detail in the lecture "Processor and programs".

Unfortunately, in practice, we face an unavoidable ambiguity of naming, with *primary memory*, *main memory* and *internal memory* used interchangeably to denote either the whole primary memory or just its largest and slowest layer. Also, the term *RAM*, while literally expanding to "random access memory (of any kind)", is commonly used to describe the main memory — and so will happen in the remaining part of this lecture.

Hardware properties

Static RAM (SRAM)

The most effective memory type is **static memory** (**SRAM**), called so because it does not need periodic refreshing of its content (unlike the dynamic RAM, DRAM).

On the other hand, SRAM is more expensive than DRAM, as a result of using more transistors per a single memory cell. This also makes SRAM less *dense* (i.e. taking physically more space). For these reasons, SRAM is used in processor registers and relatively small cache memories, but not in much larger RAM modules in personal computers.

Dynamic RAM (DRAM)

The RAM memory, clearly the largest component of primary memory, is manufactured as **dynamic RAM** (**DRAM**), and more specifically — since the early 21st century — as **DDR SDRAM**. That acronym stands for: *double data rate synchronous dynamic random-access memory*. Let us discuss the individual parts of this long name:

- *random-access memory* — this means that data can be accessed in an arbitrary (*random*) order; the opposite to this is memory with *sequential access*, which requires that data are read in some specific order;
- *dynamic* — this means that memory needs periodic refreshing its content (i.e. reading each memory cell and writing it back to the same place);
- *synchronous* — this means that the access to memory is regulated with the system clock, as opposed to *asynchronous* memory in which the access is regulated by internal memory signals;
- *double data rate* — this means that data are sent twice as frequently as the nominal clock rate would suggest — that is, on the *rising slope* as well as on the *falling slope* of the clock signal (see the lecture "Integrated circuits").

Other important properties of the SDRAM type memory are:

- *interleaving* — allows more than one read/write operation at a time;
- returning data in the form of multi-byte chunks (which is desirable because, as soon as a particular memory cell is fetched, its neighbors are also likely to be soon needed).

During its evolution, DDR SDRAM has existed in the following configurations:

- DDR SDRAM (bandwidth: from 1,600 MB/s to 3,200 MB/s);
- DDR2 SDRAM (bandwidth: from 3,200 MB/s to 8,533 MB/s);
- DDR3 SDRAM (bandwidth: from 6,400 MB/s to 19,200 MB/s);
- DDR4 SDRAM (bandwidth: from 12,800 MB/s to 25,600 MB/s);

- DDR5 SDRAM (bandwidth: from 25,600 MB/s to 57,600 MB/s).

Unlike cache, DDR SDRAM is a separate integrated circuit, which makes it possible to take it out from a computer and replace by another; often it's also possible to extend the existing RAM modules with additional ones. However, there are some technical limitations here. Physically, the memory is inserted into the *motherboard* which ensures communication between the computer components. Clearly, one cannot insert more RAM modules than the number of available memory ports. Another limitation comes from the processor, which is designed with an upper bound of supportable memory size. Also, not all processors / motherboards can support DDR4. Otherwise, a DDR4 module can be placed in a computer, though it will behave as DDR3 (which matters e.g. for the voltage input; DDR4 is capable of operating on a lower voltage, leading to reduced heat and increased performance).

The size of available RAM memory varies greatly across computers: in typical modern personal devices, it can range from a few up to hundreds of gigabytes.

An important characteristic of the processor and the motherboard is the number of **channels**. A *single-channel* architecture means that RAM has only one connection (*channel*) with the processor. *Dual-channel* means there are 2 such connections; the number of channels can be also 3, 4, or 8. Having more channels allows the processor to communicate simultaneously with more memory modules. Therefore, using e.g. two 4 GB modules will be more efficient than one 8 GB module. Benefitting from multi-channel transmission may, however, require ensuring that the modules used have the same manufacturer and/or capacity.

Cache

Generally, the concept of *cache* in computer science denotes the following situation:

- The starting point is storing data in a large though not so fast memory M — and wishing to accelerate access to these data.
- To achieve that, we introduce **cache** C, that is: a smaller buffer made of memory which allows faster access, containing those data from M which we expect to be relevant in the near future. Thus, we replicate some data stored in the original location M — though, in reward, we ensure faster reachability for them. Typically, writing to the memory is also done via this buffer.

In the case of **CPU cache**, the role of M will be played by RAM. However, it's worth noting that similar problems and solutions appear also elsewhere, including the software layer. (For example: for a large, distributed database, it often pays off to replicate some data and store them locally).

Rules of operation

Caches can generally work in various ways, depending on how we answer the following questions:

- Where in the buffer can we store the given value?
- If the buffer is filled up, how should we choose an old value to replaced by the new one?
- How long does it take to find a requested value in the cache?

- How much time do we lose in case of a cache **miss**?
  (That is related to the difference between access times for M and for C)
- In case when we do fetch a value from M, should we store in the cache just that value, or also some other values related to it?
  (An example from the database world: after retrieving a customer's last name, we may reasonably expect a need to fetch their first name — so, whenever we fetch the former, it may pay off to proactively also fetch the latter).

In the case of CPU cache, **speed** is a decision factor of utmost importance: checking for a value in the cache should take as little clock cycles as possible (and, to achieve that, we can even accept an increased risk of a cache miss). This leads to the following *simplified* rules of operation (which we will below amend to a somewhat more precise form):

- The cache is an array of 2^N **entries**, each of which contains some value from the RAM together with its source address.
- Whenever the CPU needs a value from an address A, it checks the cache only at the **index** A `%` 2^N (where `%` is the *modulo* operation, i.e. the division remainder).
- If we do fetch the value from an address A in RAM, we store it always at the index A `%` 2^N. The previous cache entry at that index is overwritten.

Such an approach is very fast (we check only one cache entry, which in addition requires very light computations, as A `%` 2^N is just the last N bits of A). However, it has its drawbacks: the risk of a miss is significantly increased (e.g. two frequently used addresses pointing to the same index would constantly expel each other from the cache); also, any cross-data relationships are disregarded.

For these reasons, in practice, CPU cache introduces the following enhancements:

- We allow an increased **set-associativity** D (typically: 2, 4, or 8), which means that each index stores D entries instead of one.
  Now, on every RAM access, we check *all* entries at the appropriate index. If any of them matches the deisred address, we have a cache hit (and can stop searching). If none of them was a hit, we claim a miss, fetch the value from RAM, and choose one of the D entries to be replaced.
  By doing this, we may slow down cache access by up to D times, in exchange for increasing the cache hit rate.
- We increase the **value size** of cache entries to 2^K bytes (typically, K is 6 or 7).
  This trick increases the chance of related data coexisting in the cache (as programmers often use data arrays, or sets of variables that had been allocated next to each other, etc.). It also decreases the fraction of cache capacity "wasted" on storing the source addresses.

  In practice: if we want to get the A-th byte of RAM, we transform A to the appropriate index of a 2^K-byte memory cell (result: "the $(A/2^K)$-th cell"), determine the appropriate index in the cache (result: $(A/2^K)$ `%` 2^N), and — if a cache hit occurred — pick the (A `%` 2^K)-th byte of the cached value.
  Note that all these address operations still boil down to very simple bit shifts or masks.
- Finally, to reduce memory footprint, we equip each cache entry no longer with the full source address but rather with its **non-obvious prefix** (omitting the last $N + K$ bits, which in practice makes up about a half of the total address length).

Cache levels

The CPU cache is split into (typically) 3 parts, called **levels** and denoted L1, L2, L3.

The **L1 cache** was already featured in the lecture about processor orders. It's here where the orders are initially decoded, and their arguments fetched. L1 cache observes a split into data part and instruction part (so that half of it can only store the actual data, and half of it — the code). Its size is modest — typically 256 kB. The set-associativity is typically 2 or 4. Each processor code has its own L1 cache. The transmission bandwidth is between 25 and 700 GB/s, and the access time is about 0.5 ns (reminder: $1\,\text{ns} = 10^{-9}\,s$).

L1 cache does not fetch data directly from RAM, but from another level, **L2 cache**. There, we lose the split into data part and instruction part, though we usually retain the split per processor core. The sizes vary greatly but usually fall between 64 kB and 12 MB. The set-associativity is typically 2, 4 or 8. The typical bandwidth is 15-200 GB/s, and typical access time is 5-10 ns.

The last level, **L3 cache**, is largest and slowest. Its size is typically 4-64 MB, and set-associativity is 16 or more. The bandwidth is 10-100 GB/s, and access time is typically 20-50 ns. This type of cache is shared across processor cores, which makes it crucial for synchronizing their work.

In some processors, there is also **L4 cache**, manufactured in the technology called **eDRAM** (*embedded DRAM*). That type has a higher memory density than SRAM, and is almost equally expensive. Despite being a dynamic memory (so requiring periodic refreshing), the technical details (integration with the processor and equipping with a dedicated controller) make the refreshing less visible for the user, to the point where eDRAM can be considered as if it was some simple kind of SRAM. Moreover, placing L4 cache inside the processor allows to use a more efficient transfer bus.

Addressing RAM memory

We will now discuss the topic of memory addressing. As explained on the lecture "On the assembler", processes are equipped with their private pools of RAM, to which they refer with the so-called **logical addresses**. The programmer (or the compiler) uses the logical addresses as if the first memory cell had address 0, and the whole area of RAM assigned to the process was contiguous. We will now discuss two techniques of address translation: **segmentation** and **paging**.

Segmentation

We have already came across segmentation on the lecture "On the assembler".

The memory assigned to a process is split into a few **segments**, whose specifications (starting address, size, access mode) are stored in the **segment descriptor table**. The **virtual address** — the one which can be used by an assmebler programmer — consists of two parts:

- the **segment selector** (specifying that the desired value belongs e.g. to the code segment)
- the **offset**, i.e. the index of the desired memory cell within that segment.

To obtain the **logical address** corresponding to a given virtual address, we need to look up the base address of the selected segment in the segment descriptor table, and then add the offset to it.

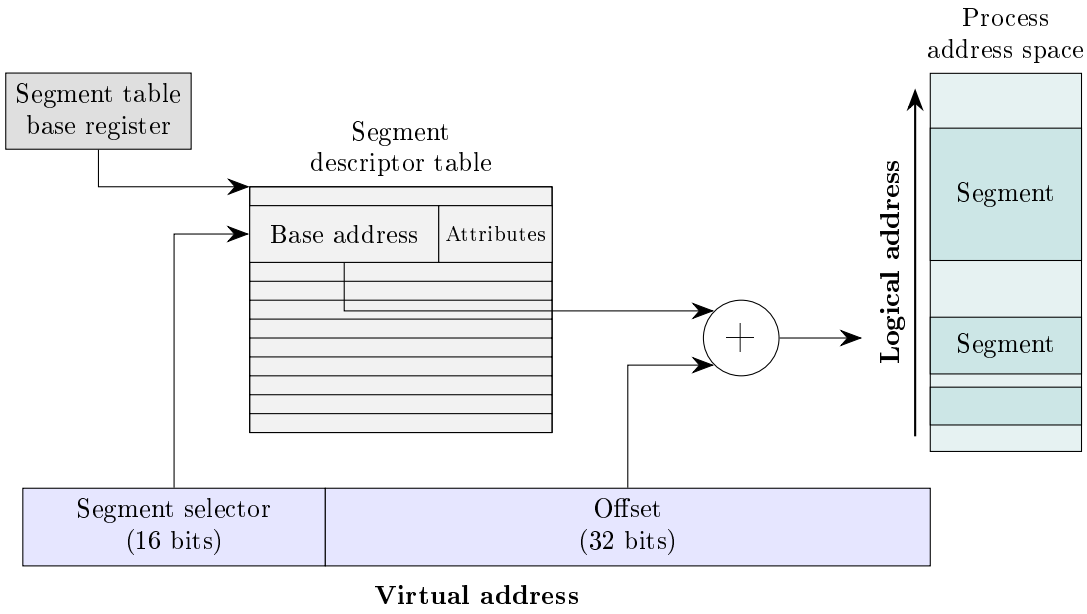

Figure 1. Segmentation in the Intel 80386 processor.

At this point, however, we need to make three disclaimers:

- Translating a virtaul address to a logical one used to happen in the way described above in the 32-bit processors from the x86 family. In the newer 64-bit version known as **x86-64**, in the 64-bit modes, segmentation becomes reduced (though still present to some extent).
- In the assembler code, the segment selector is often implicit as the compiler can deduce it from the context. (For example: the target addresses for jump instructions are interpreted within code segment; the data pointers within data segment, etc.)
  If the programmer needs to specify the segment selector explicitly, they can use an appropriate extended syntax, e.g. `cs:eax` means "an address with offset specified by the value of `eax`, and the code segment register value (`cs`) used as the segment selector".
- Again, the terminology commonly used to describe segmentation contains many divergences. For example:
  - the phrases *logical address* and *virtual address* happen to be used interchangeably; the logical address is also sometimes called the *line address*;
  - the word *segment* itself can mean an area of data described by a segment descriptor (like above), but also a functional subset of a process address space (e.g. the heap, stack etc.), or even an area of memory (with specific access mode) assigned to a process by the operating system (which does not need to coincide with a "segment" from the processor viewpoint).

Paging

Let us recall that the logical address should not be confused with the **physical address**, which specifies the actual placement of data in RAM. After all, as every process uses its own logical addresses between 0 and 4 GiB, various processes will generally operate on the same logical addresses, corresponding to different physical addresses. To translate the logical address into the physical one, we use a **page table** for the current process, maintained and stored by the operating system. The basic idea of translation using paging looks as follows:

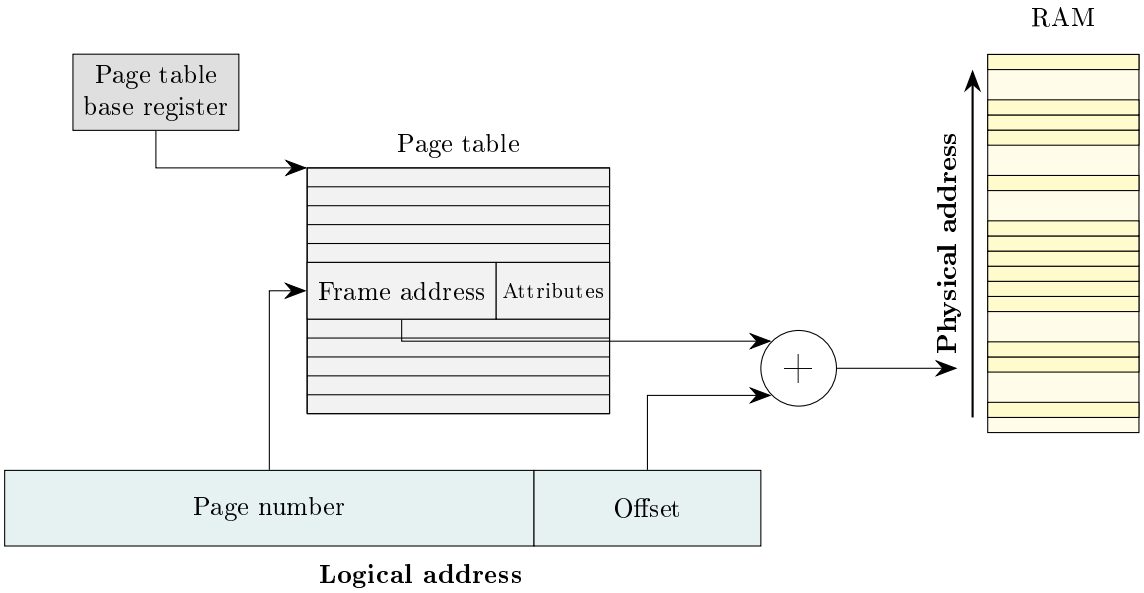

Figure 2. The overall idea of paging.
Although the general principle seems very similar to the picture for segmentation (see Figure 1), the main difference is in the equal sizes of all RAM frames, which allows unconstrained bookkeeping. (Another important difference, which we will discuss later in this lecture, is the *virtualization* technique).

What can be seen here is a split of the process address space into **pages** of a fixed size (typically 4 kiB), and the split of RAM into **frames** of the same size.

By introducing a fixed page size, the operating system can grant memory to processes according to their changing needs, assigning individual frames to individual processes (and to some logical addresses in their address spaces) in a *completely unconstrained* manner. This removes the problem known as **external memory fragmentation**, i.e. the scenario in which the free space in RAM memory gets split into so many smaller parts that some requests for a contiguous memory block cannot be served even though the request block size is smaller than the total amount of free memory. To the contrary, paging ensures that, as long as free memory exists at all, it is available in the form of free frames, and moreover, free frames have a suitable format for serving memory requests.

On the other hand, paging brings another kind of problem, the **internal fragmentation**: the amount of memory granted to a process is equal to its actual need but rounded up (to a whole number of pages); therefore, the rounding margin is effectively wasted. Therefore, choosing the page size involves a trade-off between reducing internal fragmentation (for which we'd prefer smaller pages) and reducing memory footprint of the page table (for which larger pages are better).

Multi-level paging

A single-level page table turns out to be an overly simplistic technique, especially when we look at its memory consumption. If the address space of a process (4 GiB) is split into pages of size 4 kiB, and each page descriptor takes e.g. 24 bits, then the page table for a single process will take 3 MiB, and all the tables for all currently running processes — often over 1 GiB! In such case, a common solution is to apply **multi-level paging**, in which the page table itself is subject to additional paging. For example, two-level paging proceeds as shown below:

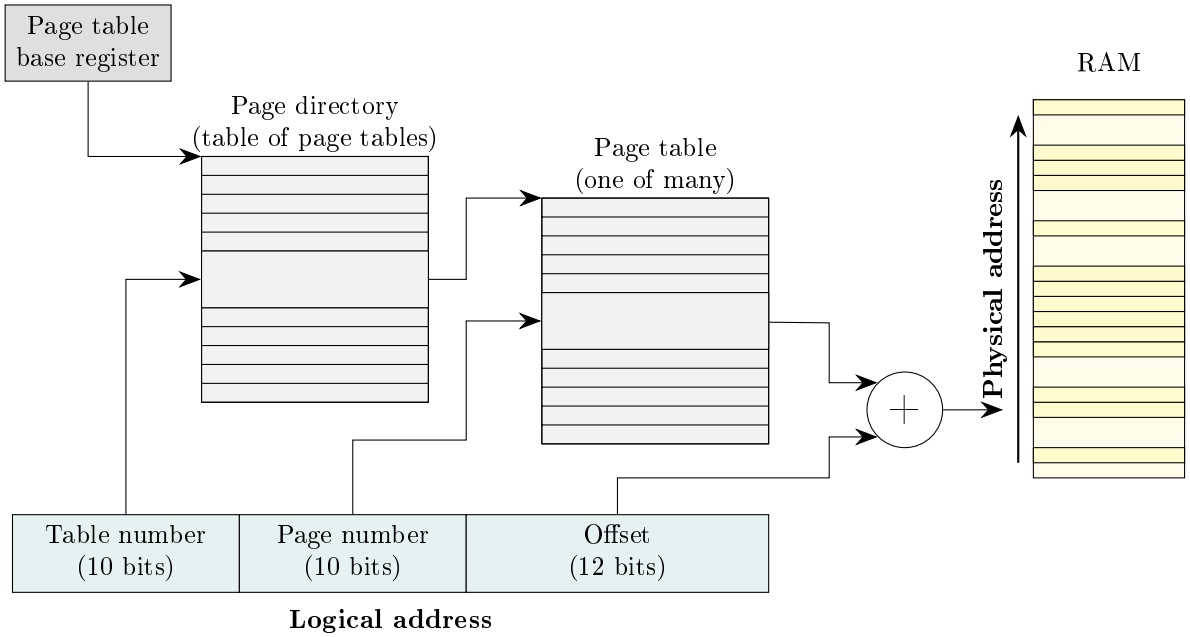

Figure 3. Two-level paging in the Intel 80386 processor.

A disadvantage of this solution is the increase of access time: by using n-level paging, reaching the desired RAM address may require altogether $n + 1$ read operations from that memory. (Generally, page tables are placed in RAM; however, due to how frequently they are referenced, processors contain a dedicated cache for them, called **TLB**, *translation lookaside buffers*). However, the main advantage of multi-level paging is more efficient management of memory used for page tables: those not used might not exist at all, and those used rarely can be safely sent back to secondary memory (see below).

The modern x86 processors use **4 or 5 levels** of paging.

Virtualization

One advantage of paging already known to us is solving the problem of external memory fragmentation. However, paging also brings additional opportunities, among which the crucial one is the so-called **memory virtualization**. This technique allows *referring to* a total amount of RAM memory (by which we mean the total size of address spaces of all processes) **larger than** its physical capacity! Such "cheating" is possible thanks to the fact that the frames of data which do not fit into RAM are in fact stored in the secondary memory (e.g. on the hard drive) and brought back to RAM when needed.

In the virtualized setup, fetching data from RAM engages the processor MMU as well as the operating system. To decide where the data should be fetched from (RAM or hard drive?), we use additional information stored in page descriptors. These contain in particular the **present bit**, specifying whether the page is available in RAM. If not (which means it should be retrieved from the hard drive), the processor generates a **page fault interrupt**, bringing control to the operating system, which manages fetching the appropriate frame from the hard drive to RAM. After such fetching, the page is now placed in the main memory, so we need to update its descriptor in the page table (in particular, store the new physical address, and flip the present bit).

This leads to another problem: when fetching a new page to the RAM, *where* should it be placed? It must replace another page, and clearly, the best one to replace would be one that is not going to be used anymore, which may exist for various reasons (e.g. because another process has just finished

executing, or because its local data are no longer live, e.g. as a result of freeing previously allocated memory). To allow detecting such cases, page descriptors contain the **use bit** controlled by the operating system.

However, the operating system will not always find an unused location in the RAM. If there's no free space while we fetch a new page, we must pick another *least necessary* page to be sent back to the hard drive. The problem of choosing a *least necessary* page is complex and can be solved with various techniques. The following are the simplest strategies for this:

- **FIFO** (*First In First Out*) — always send back the page which has been fetched longest time ago;
- **LRU** (*Least Recently Used*) — send back the page which has been used longest time ago;
- **LFU** (*Least Frequently Used*) — equip each page with a *reference counter*, initialized when the page is fetched and incremented on every reference to that page; send back the page for which the reference counter has the lowest value.

The above paragraph contains multiple simplifications but it (hopefully) shows the main idea behind memory virtualization. Unfortunately, describing the details of choosing pages and exchanging them between the hard drive and RAM is out of scope of this course.

Virtualization vs. caching

Note that the main idea of caching — moving data between a larger slower memory (M) and a smaller faster one (C) — does take place also in case of virtualization: this time, RAM plays the role of C, and M is the secondary memory. Yet, despite this apparent similarity, virtualization *is not* a special case of caching.

The purpose of caching is *accelerating* the access to memory M, by introducing C as a buffer. This affects just the efficiency; no cache would mean working just with M, which would be slower but would still work. The case of virtualization is different: the data in M (on a hard drive) are *useless* unless they are fetched to smaller memory C (RAM), and the goal of the whole mechanism is to improve *scalability* of the system (i.e. enable executing more processes at a time, and assigning a larger total amount of memory to those processess) at the cost of efficiency.