Distinct benefits of virtual memory:

1. **multiprogramming**: Switch between processes without reloading memory.

2. **protection**: Instructions run by one process cannot access data belonging to a different process.

3. **relocation**: Final (runtime) physical addresses are not known when the program is linked.

4. **swapping/paging**: Programs can be written as if their memory is larger than physical memory because the pieces of program memory can be stored on a disk.

Swapping is an example of **caching**: An expensive resource is created ahead of time and kept safely for future reuse.

**Memory Hierarchy** managed by hw/sw.

Historically, each benefit had been realized by different mechanisms: partitions, prot. keys, base/limit regs, patching, position independent code, overlays, swapping **entire processes**.
Suppose a process on a \textbf{dedicated} computer spends 20\% of its time computing; 80\% of the time it’s waiting for I/O.

Crude analysis: 5 such processes timeshared will fully utilize the CPU.

Better model: In a random snapshot of the dedicated computer, 
\begin{equation}
\text{prob. (the process is waiting for I/O)} = 0.80 = p
\end{equation}

With \( n \) similar processes running in \textbf{independent} computers
\begin{equation}
p^n = \text{prob. (All } n \text{ processes are waiting for I/O)}
\end{equation}
\begin{equation}
1 - p^n = \text{prob. (At least one is using the CPU)}.
\end{equation}

This is a good approximation for \textbf{the CPU utilization} as a function of the \textbf{degree of multiprogramming} \( n \), under the simplification that we can ignore the time during which \( k > 1 \) processes are ready (so \( k - 1 \) of them are waiting for the CPU, not I/O). Can now predict:

\begin{equation}
\text{CPU work on one job} = \frac{\text{elapsed time}}{\text{deg. of multiprogramming}} \times \text{CPU utiliz.}
\end{equation}
Memory management in Linux:

1. Application of ia32 (hardware) segments, pages, their descriptors, control registers, etc. to Linux addressing concepts. [ULK Chap. 2]

2. Mgt. of paged memory used for a *variety* of data structures within the kernel and for process’ virtual memory. [ULK Chap. 6]

3. Segmentation of process virtual address spaces by use of virtual address intervals (not ia32 hardware segment features). Page fault handling. (Pages for data structures *within* the Linux kernel are not swapped.) [ULK Chap. 7]

4. **Swapping out** of pages belonging to processes. (Kernel pages are not swapped in Linux.) [ULK Chap. 16]
   Also, buffered and memory mapped files. [ULK Chap. 15]
Virtual Memory implemented by paging:

Memory Management Unit (hardware) maps virtual addresses into physical addresses (and causes a page fault sometimes).

Allocated virtual address space (Unix Segments) is divided into units of pages (usually equal in size.) “Page” also denotes the unit’s contents.

Physical memory is divided into units of page frames.

Basic memory management data structure tells which allocation units are in use.

Demand Paging: Pages are not allocated until the process begins to run, thence only pages used are ever loaded.

Modern systems load executable and library files that way, with memory mapped files

Implementations: bitmap; ordered linked list;

combinations in Linux: array with some doubly linked entries; balanced search tree of doubly linked nodes.
The MMU consists of a Translation Lookaside Buffer plus (in some ISAs such as ia32, not MIPS) hardware that consults the in-memory page tables. (Most) hardware caches translations found in page tables into the TLB. (In systems like MIPS, the OS uses software page tables to load the TLB after TLB exceptions). So, most memory references DON’T cause page table access.

If it weren’t for TLB technology, paging would be impractical.
When the current process tries to access an address that is not in a **present** page, the OS must copy that page from the disk (or create an empty page). It needs a **free page frame**.

**Page Replacement Algorithms:** Strategies to choose a page to **evict** when an unused page frame is needed, or to ensure a future supply of unused page frames.

**Dirty Page:** A page whose contents are not backed **up on the disk**. *Hardware* sets the PTE’s M bit when a page becomes dirty. It must not be evicted until its contents are saved!

One strategic idea is for a **paging daemon** to **concurrently** (Sec. 4.6.6) write back dirty pages (make them **clean**) before free page frames are actually needed. OS resets the M bit. [A caching strategy!!]

**General cache idea:** A resource with expensive creation operations is kept so its future uses are cheap. EG: (1) access to memory locations through HW caches, (2) access to VM pages, (3) free page frames, (4) free buffers or other data structures.
Optimum Page Replacement Algorithm: Evict the page that will be used last. This is impossible to predict!

Not Recently Used: Periodically classify pages based on their hardware set R(Referenced) and M(Modified) bits. Choose randomly from the lowest numbered non-empty class:
Class 0: not R, not M
Class 1: not R, is M (evicting this is more useful)
Class 2: is R, not M
Class 3: is R and is M

First-In First-Out: Next evicted page is the page that was in memory the longest. Not very good. Subject to Belady’s anomaly.

Second Chance aka clock: Modified FIFO so if the first had been referenced, update its to position to last and reset its R bit.

The clock implementation avoids re-ordering the list of pages in order of age.
Several strategies below are based on *time* since last access to a page rather than order of creation or access.

**Least Recently Used:** Optimal is impossible to implement. But locality of reference makes LRU a good approximation: What predicts what??

LRU is not practical, but useful in simulation studies.

**Not Frequently Used:** For all pages, OS counts R bit settings for all pages during previous clock period. Practical??? Also doesn’t forget the past. (Not time based.)

**Aging:** Count = $\alpha \ast \text{Count} + 1$

Book shows $\alpha = 1/2$. See aging of CPU time used in scheduling too.
Some strategies try to estimate what pages are in each process’s **Working Set**

\[ W(k, t) = \text{set of pages used by the } k \text{ most recent memory references, defined at virtual time } t. \]

Note that \(|W(k, t)| \leq k\) and it’s usually \(<< k\) since each page is usually referenced *repeatedly*.

Typical working set size \(|W(k, t)|\) behavior graph (as a function of \(k\)) Figure 4-20.

**Thrashing** occurs if \#pages available \(< |W(k, t)|\) for too small \(k\).
Working Set Replacement:

Ideas:

Approximate $W(k,t)$ by the set of pages referenced during the last $\tau$ msec of virtual time.

Maintain the approx. virtual time of last use “TLU” in the page’s information structure.

Approximate this by sampling: only during a page fault, update the TLU for pages referenced during the last tick.

Algorithm outline:

(Assume the R bits are cleared at each tick.)

After a page fault, scan every page:

If $R==1$ then update time of last use.

If $R!=1$ then calculate virtual age (how?) and compare to $\tau$.

If virtual age $> \tau$ evict the page.

Otherwise, update record of oldest page.

If no unreferenced pages are found, evict a randomly chosen referenced page, preferably clean.
**WSClock Replacement:** Like WSR but avoids scanning entire page table.

Uses the clock algorithm’s circular linked list of pages and WSR’s virtual TLU field.

At each page fault, advance the “hand” until a clean page older than $\tau$ is found.

If $R==1$ then reset $R$, update TLU, continue.

If $R==0$, age $> \tau$, and page is clean, **done**.

If $R==0$, age $> \tau$, and page is dirty, *schedule a write* (up to a limit, perhaps) and continue.

After one time around, if writes were scheduled, continue.*Why is that OK?* if no writes scheduled (so all pages are “young”), evict any clean page; failing that, write back (make clean) and evict the current (formerly dirty) page.