Computer Science 432/563
Operating Systems
The College of Saint Rose
Spring 2016

Topic Notes: Memory Management

Memory is one of the major shared resources in any modern computer system. A program needs access to the CPU and space in memory for its instructions and data in order to run.

Think of a simple program that does some operations like this:

```
  x = x + 1;

  // stuff

  do {

    // stuff
  while (i!=0);
```

This becomes assembly code that looks something like this:

```
  LOAD X, R0
  INC R0
  STOR R0, X
```

and

```
loopstart:
  ...
  ! assuming i is in R1
  TEST R1
  BNE loopstart
```

Somewhere in memory, there is program text corresponding to these statements and variables.

How does the program know how to find the memory location corresponding to variable `x`? How does the `BNE` instruction know where to jump to?

Binding to Memory

When does an instruction or a variable get assigned an actual memory address? There are three possible times:

1. *Programming or Compile time*: If we can know the actual memory location *a priori*, absolute code can be generated. The downside is that we must rewrite or recompile code if its starting location changes.

   This might be used on small systems.

   - Microprocessors might do this.
   - The old DOS `.com` format programs used this.
   - Programs for things like the Commodore 64 used this.

   For example, a BASIC program on the Commodore 64 could include the statements

   ```
   10 POKE 1320, 1
   20 POKE 55592, 6
   ```

   This puts character 'A' near the middle of the top of the screen, then changes its color to blue. (See Appendix D of the *Commodore 64 Programmer's Reference Guide*)

   We could also see the current status of joystick 1:

   ```
   10 JV=PEEK(56320)
   ```

   Bits correspond to the 4 directions and the fire button status.

   A program could also decide to `POKE` and `PEEK` values anywhere into memory.

   This will not work on a multiprogrammed system, unless each program is compiled to have disjoint memory usage, but the idea lives on in small devices today.

2. *Load time*: Must generate relocatable code if memory location is not known at compile time.

   Medium/larger systems may do this. The physical address for a variable or an instruction (branch target) is computed when the program is loaded into memory.

   Once the program has been loaded into a section of memory, it cannot move. If it is removed from memory, it must be returned to the original location.

   Multiprogramming is possible, and we will consider the issues this brings up for memory management.

3. *Execution time*: Binding must be delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base and limit registers*).

   This is also used for modern medium and large systems. A program's data and instructions can be moved around in memory at run time.

   This allows *dynamic loading* and/or *dynamic linking*. Here, a segment of program code is not brought into memory until it is needed.

Logical vs. Physical Address Space

A *logical address* is one generated by the CPU (your program); and is also referred to as *virtual address*.

A *physical address* is the one seen by the memory unit.

These are the same in compile-time and load-time address-binding schemes; they differ in execution-time address-binding scheme.

The *Memory-Management Unit (MMU)* is a hardware device that maps virtual to physical address. In a simple MMU scheme, the value in the *relocation register* is added to every address generated by a user process at the time it is sent to memory.

The user program deals with logical addresses; it never sees the real physical addresses.

If multiple processes are going to be in memory, we need to make sure that processes cannot interfere with each others' memory. Each memory management scheme we consider must provide for some form of *memory protection*.

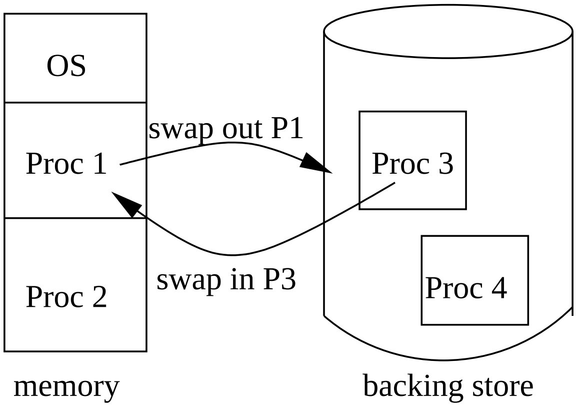
Swapping

In a multiprogrammed system, there may not be enough memory to have all processes that are in the system in memory at once. If this is the case, programs must be brought into memory when they are selected to run on the CPU. This is where *medium-term scheduling* comes in.

Swapping is when a process is moved from main memory to the *backing store*, then brought back into memory later for continued execution.

The backing store is usually a disk, which does have space to hold memory images for all processes.

Swapping time is dominated by transfer time of this backing store, which is directly proportional to the amount of memory swapped.



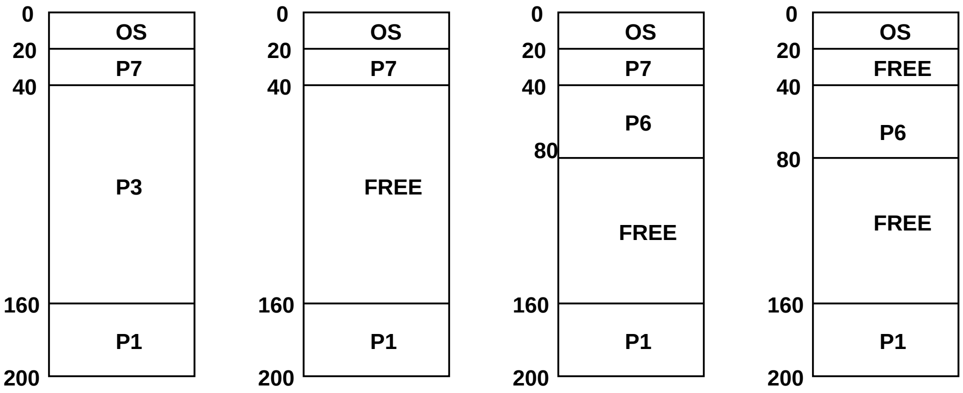
The short-term scheduler can only choose among the processes in memory. So to keep the CPU as busy as possible, we would like a number of processes in memory. Assuming than an entire process must be in memory (not a valid assumption when we add virtual memory), how can we allocate the available memory among the processes?

Contiguous Allocation

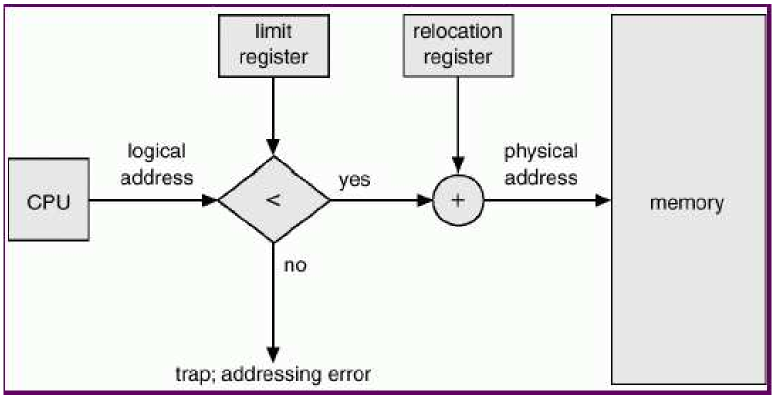
If all processes are the same size (not likely!), we could divide up the available memory into chunks of that size, and swap processes in and out of these chunks. In reality, they are different sizes.

For the moment, we will assume that each process may have a different size, but that its size is fixed throughout its lifetime. Processes are allocated chunks of contiguous memory of various sizes.

When a process arrives (or is swapped in) it is allocated an available chunk (a "hole") large enough to hold it. The system needs to remember what memory is allocated and what memory is free.



What hardware support is needed to support this relocation?



CPU's logical address is first checked for legality (limit register) to ensure that it will be mapped into this process' physical address space, then it is added to an offset (relocation or base register) to get the physical address. If the program is reloaded into a different portion of memory later, the same logical addresses remain valid, but the base register will change.

How do we decide which "hole" to use when a process is added? We may have several available holes to choose from, and it may be advantageous to choose one over another.

- *First-fit*: choose the first hole we find that is large enough. This is fast, minimizing the search.
- *Best-fit*: Allocate the smallest available hole that is large enough to work. A search is needed. The search may be shortened by maintaining the list of holes ordered by size.
- *Worst-fit*: Allocate the largest hole. This is counterintuitive, but may be reasonable. It produces the largest leftover hole. However, in practice it performs worse.

Fragmentation

The problem with any of these is that once a number of processes have come and gone, we may have shredded up our memory into a bunch of small holes, each of which alone may be too small to be of much use, but could be significant when considered together. This is known as *fragmentation*.

- *External fragmentation*: total memory space exists to satisfy a request, but it is not contiuguous. For example: 3 holes of size 20 are available, but a process cannot be allocated because it requires 30.
- *Internal fragmentation*: this occurs when the size of all memory requests are rounded up to the next multiple of some convenient size, say 4K. So if a process needs 7K, it needs to round up to 8K, and the 1K extra is wasted space. The cost may be worthwhile, as this could decrease external fragmentation.

External fragmentation can be reduced by *compaction* – shuffling of allocated memory around to turn the small holes into one big chunk of available memory. This can be done, assuming relocatable code, but it is expensive!

Contiguous allocation will not be sufficient for most real systems.

More Advanced Approaches

Contiguous allocation is unlikely to work in most environments. The most common approaches used today are *paging*, *segmentation*, or a combination of these.

In each case, we break down the logical address space for a process into smaller chunks. It is these chunks that we assign to parts of physical memory rather than the whole process.

With paging, we break up memory into fixed-size chunks. With segmentation, we break up memory into pieces whose size corresponds to some logical memory division of the program, such as a block of global variables, the program text for a particular function, etc.

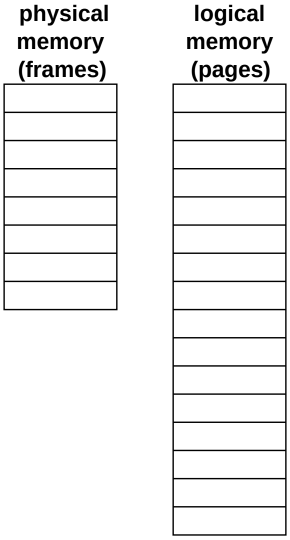
Paging

We can do better if we allow a process' logical memory to be noncontiguous in physical memory. An approach that breaks up memory into fixed-size chunks is called *paging*.

The size of the chunks is called the *page size*, typically a power of 2 around 4K. We break up both our logical memory and physical memory into chunks of this size. The logical memory chunks are called *pages* and the physical memory chunks are called *frames*.

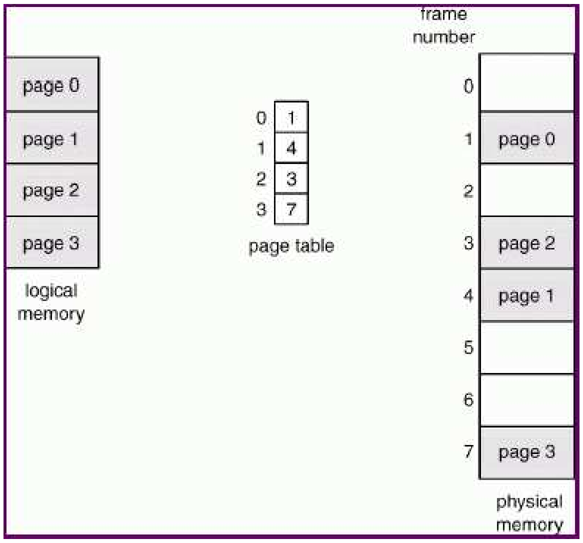
The system keeps track of the free frames of physical memory, and when a program of size n pages is to be loaded, n free frames must be located.

We can create a *virtual memory* which is larger than our physical memory by extending the idea of process swapping to allow swapping of individual pages. This allows only part of a process to be in memory at a time, and in fact allows programs to access a logical memory that is larger than the entire physical memory of the system.



Fragmentation: we have no external fragmentation, but we do have internal fragmentation.

The contiguous allocation scheme required only a pair of registers, the base/relocation register and the limit register to translate logical addresses to physical addresses and to provide access restrictions. For paging, we need something more complicated. A *page table* is needed to translate logical to physical addresses.



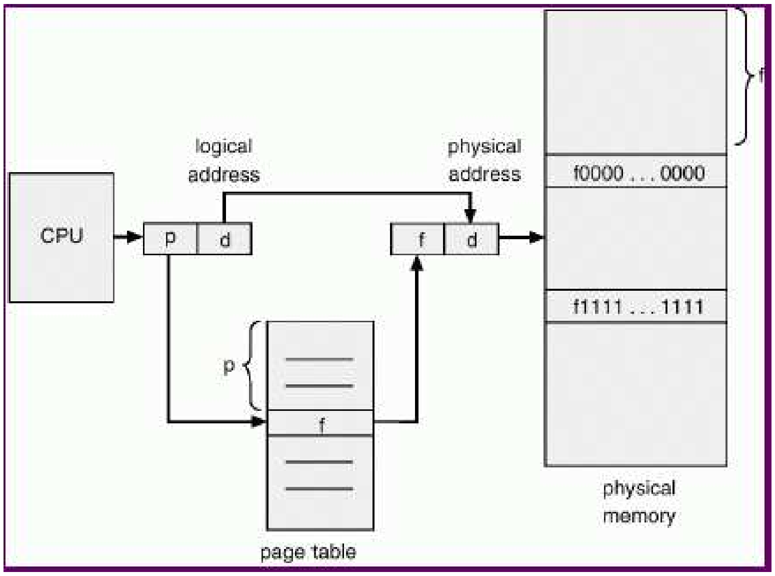
A page table is maintained for each process, and is maintained in main memory (in its most straightforward implementation).

A page table base register can be used to locate the page table in memory, page-table length register to determine its size.

It is possible that not all of a process' pages are in memory, but we will not consider that just yet.

Logical addresses are broken into:

- A *page number*, p, which is the index into the page table
- A *page offset*, d, which is added to the base address of the physical memory frame that holds logical page p

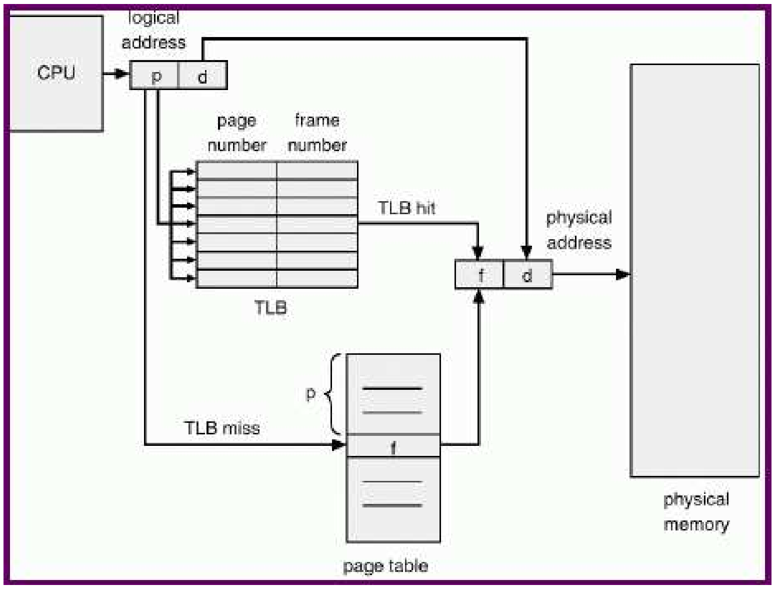


If a page moves to a different frame, we don't have to "tell" the program – just update the page table and the same logical addresses will work.

The number of bits of the address that make up p and d are determined by the page/frame size. The size is always a power of 2, and a k-bit page offset size indicates a page size of 2^k. The remaining bits of the logical address determine how many pages of logical memory can be addressed by a process. If there are j bits in the page number, then there are 2^j addressable pages.

Disadvantage: every data/instruction access now requires *two* memory accesses: one for the page table and one for the data/instruction. This is unacceptable! Memory access is slow compared to operations on registers!

The two memory access problem can be solved with hardware support, possibly by using a fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)



The TLB is an associative memory – a piece of hardware that lets us search all entries for a line whose page number is p. If it's there, we get our frame number f out without a memory access to do a page table lookup. Since this is relatively expensive hardware, it will not be large enough to hold all page table entries, only those we've accessed in the recent past. If we try to access a page that is not in the TLB, we go to the page table and look up the frame number. At this point, the page is moved into the TLB so if we look it up again in the near future, we can avoid the memory access.

Fortunately, even if the TLB is significantly smaller than the page table, we are likely to get a good number of TLB "hits". This is because of *locality* – the fact that programs tend to reuse the same pages of logical memory repeatedly before moving on to some other group of pages. This idea will come up a lot for the rest of the semester. But here, it means that if the TLB is large enough to hold $\frac{1}{3}$ of the page table entries, we will get much, much more than a $\frac{1}{3}$ hit rate on the TLB.

TLB hit is still more expensive than a direct memory access (no paging at all) but much better than the two references from before.

A TLB is typically around 64 entries. Tiny, but good enough to get a good hit rate in practice.

TLB management could be entirely in the MMU, but often (on RISC systems like Sparc, MIPS, Alpha) the management is done in software. A TLB miss is trapped to the OS to choose a TLB victim and get the new page table entry into the TLB.

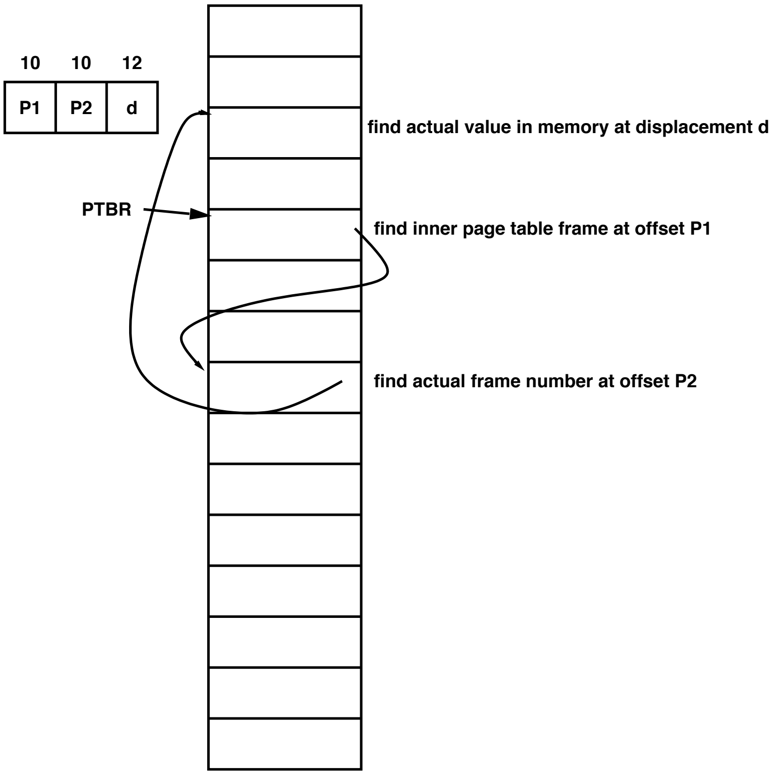
On the Intel Pentium 4 1.4 GHz (I know, ancient): 128 entries in intstruction TLB, 64 entries in data TLB. ITLB replacement takes 31 cycles, DTLB takes 48 cycles. TLBs are fully associative and use LRU replacement. (document formerly at: `http://i30www.ira.uka.de/research/documents/l4ka/smallspaces.pdf`)

Multilevel Page Tables

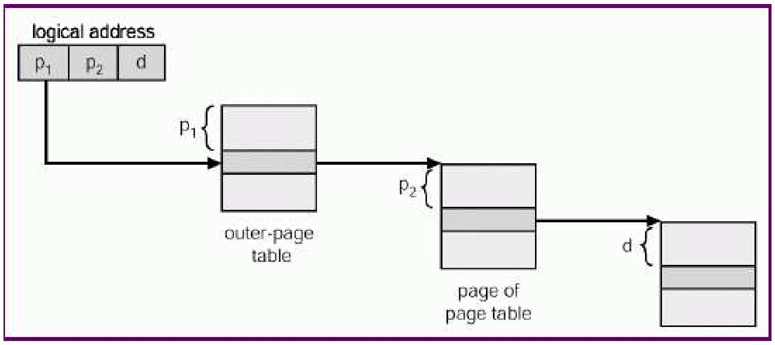
We said earlier that the page table must be kept in main memory. With a large page table, this could end up taking a significant chunk of memory.

For a 32-bit address space (2^{32} bytes) and a frame size of 4KB (2^{12}). This leaves about 1 million page table entries (2^{20}). So we use up 4 MB of memory just to hold the page table.

A solution to this is to page the page table! For a two-level paging scheme, we break up our logical address into three chunks: an index into the outer page table p_1, a displacement in the outer page table p_2 to find the actual frame, and a page offset d.


10
10
12
P1
P2
d
PTBR
find actual value in memory at displacement d
find inner page table frame at offset P1
find actual frame number at offset P2

This means our address translation is now more complex:



Note that we now have 3 memory accesses to do one real memory access! We'll need hardware support here, too.

This can be expanded to 3 or 4 levels.

This approach has been taken with some real systems: VAX/VMS uses 2 levels, Sparc uses 3 levels, and the 68030 uses 4 levels.

Think of this like a phone book (with numbers as keys to names). You don't want a big phone book with all phone numbers. You have one book which lists all area codes and tells you where to find another book with that area code. Then you find that book which lists exchanges and tells you where to find the phone book for that exchange. Finally, you look in that book for the actual numbers and find the names.

We'll really need a very high TLB hit rate to be able to make use of multilevel paging and still have reasonable performance.

If a 64-bit system wanted to use this scheme, it would take about 7 levels to get page tables down to a reasonable size. This alone is not good enough, so other approaches are needed.

Inverted Page Table

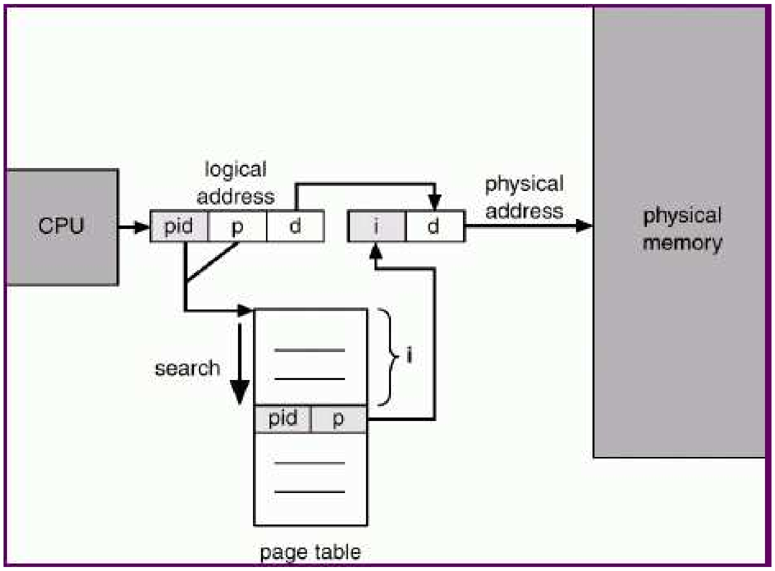
Here, we have one entry for each real page of memory, and the page table lists the pid and logical page.

An entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

This decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

A hash table can limit the search to one or at most a few page-table entries.

The extra memory accesses required for hash-table searches are alleviated by a TLB.



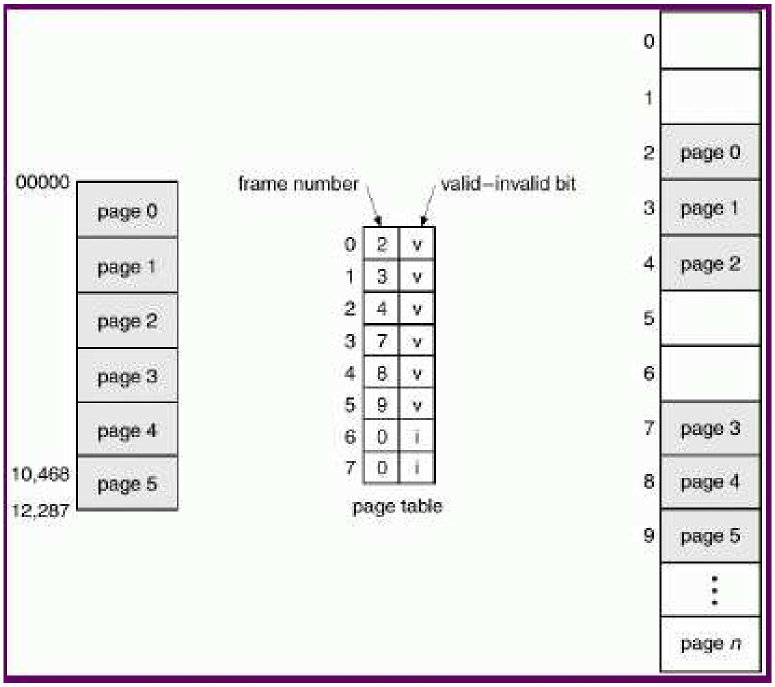
Note that there is one page table shared among all processes. The PID must be part of the logical address.

Memory Protection with Paging

Memory protection with a paging scheme is not as simple as a base/limit pair of registers.

We need to know which process owns each physical memory frame, and ensure that page table lookups for a given process can produce only physical addresses that contain that process' data.

Not all parts of the process' logical address space will actually have a physical frame assigned to them. So page table entries can be given an extra "valid-invalid" bit:



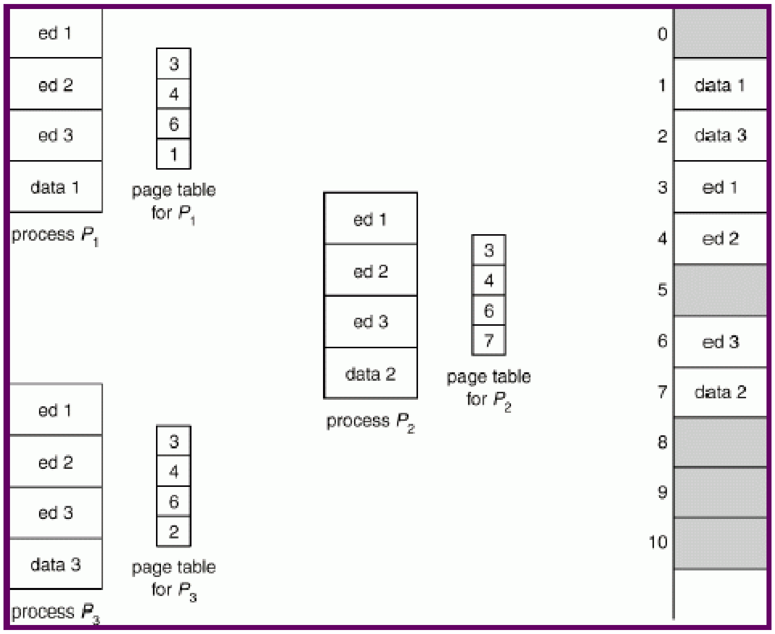
Shared Pages

Processes may be able to share code frames, and processes may wish to share memory.

- *Shared code* – Only one copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems). Such shared code must appear in same location in the logical address space of all processes.
- *Private code and data* – Each process keeps a separate copy of the code and data. The pages for the private code and data can appear anywhere in the logical address space.

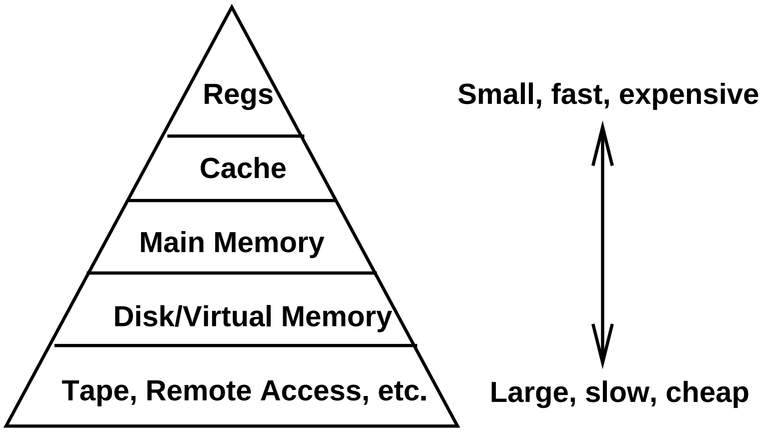
Complications:

- If a page moves, all processes sharing that page must be aware of it.
- A frame is not available for reuse until all processes that are using it no longer want it.



Demand Paging

Recall that virtual memory allows the use of a backing store (a disk) to hold pages of process' logical address space that are not currently in use.



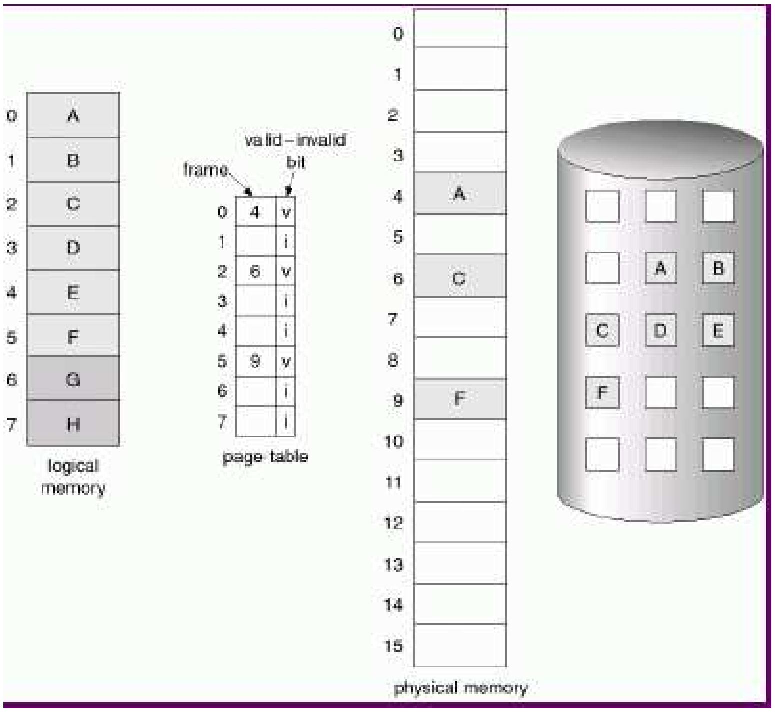
Earlier we talked about moving entire processes between virtual memory and main memory. However, when combined with a paging scheme, we can swap individual pages. This is desirable since relatively small parts of a program text and variables are active at any given time (or possibly, ever).

- Bring pages into memory only when needed
  - Less I/O needed (disk access to read pages)

- Less memory needed (physical)
- Faster response (less I/O, copying to activate a process)
- More users (more frames to go around)

- We know that a page is needed when there is a reference to it

This can be implemented with the valid-invalid bits of a page table that we saw earlier.



In the figure, logical pages A-F have been allocated by the process, while G and H are part of the logical address space but have not yet been allocated. A, C, and F, have valid page table entries, while the others do not.

If a page lookup hits a page table entry marked as invalid, we need to bring that page in. This is called a *page fault*.

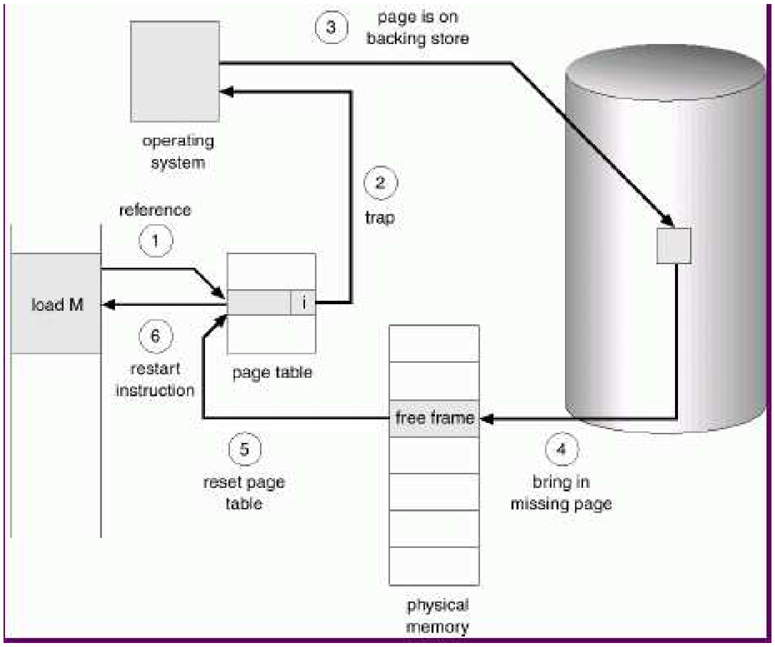
A page fault causes a trap to the OS.

- invalid reference ⟶ abort
- valid, but not-in-memory ⟶ bring to memory

If the reference is valid, the system must

- get an empty frame

- swap the page from virtual memory into the frame
- update the page table, set the status to valid
- restart the process



But what happens if there is no free frame? We need to make one!

Page Replacement

To do this, we need to take one of the frames and swap it out – send it back to the disk.

Ideally, we would like to select a frame to remove (a "victim") that is no longer in active use by any process. We will examine some algorithms that attempt to do just this. If the victim is needed again, a page fault will be generated when it is referenced and it will be brought back in.

The percentage of memory accesses that generate a page fault is called the *page fault rate*. $0.0 \leq p \leq 1.0$ A rate of 0 means there are no page faults, 1 means every reference generates a page fault.

Given a page fault rate p, memory access cost t_{ma}, and a page fault overhead t_{pf}, we can compute an *effective access time*:

$$EAT = (1 - p) \times t_{ma} + p \times t_{pf}$$

t_{pf} includes all costs of page faults, which are time to trap to the system, time to select a victim, time to swap out the page, time to swap in the referenced page, and time to restart the process.

The page being swapped out needs to be written back to disk only if it has been modified (often referred to as a "dirty" page). If the page has not been modified since we brought it into main memory, we may be able to skip this step. This, however, depends on the implementation.

For example, suppose a memory access takes 1 μs, trap to the system and selection of a victim takes 50 μs, swapping in or out a page takes 10 ms ($= 10000 \mu s$), and restarting the process takes 50 μs. Also, suppose that the page being replaced has been modified 50% of the time, so half of the page faults require both a page write and a page read.

$$t_{pf} = 50 + 5000 + 10000 + 50 = 15100$$
$$EAT = (1-p) \times t_{ma} + p \times t_{pf} = (1-p) \times 1 + p \times 15100 = 1 + 15099p$$

If $p = .5$, this is pretty horrendous. Our demand paging system has made the average memory access 7500 times slower! So in reality, p must be very small. Fortunately, our friend *locality of reference* helps us out here again. Real programs will have a low page fault rate, giving reasonable effective access times.

Appropriate selection of the victim can make a big difference here. It may be worthwhile to take more time to select a victim more carefully if it will lower p, and in turn, EAT.

To reduce the page fault rate, we should select victims that will not be used for the longest time into the future.

Page Replacement Algorithms

We were discussing memory management in paged systems.

Much like what we did to compare CPU scheduling algorithms, we will evaluate page replacement algorithms by running them on a particular string of memory references (reference string) and computing the number of page faults on that string. The string indicates the logical pages that contain successive memory accesses. We count the number of page faults needed to process the string using the given algorithm.

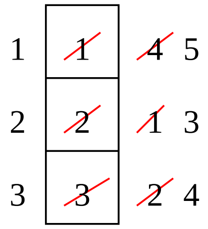
In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

We know that there must be a minimum of 5 page faults here, as 5 different pages are referenced. If we have 5 frames available, that is all we'll ever have. So we consider cases where we have 3 or 4 frames available, just to make it (slightly) interesting.

First-In-First-Out (FIFO) Algorithm

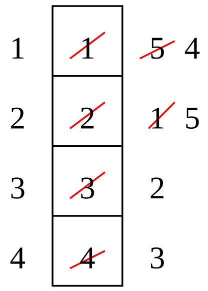
You guessed it – the first page in is the first page out when it comes time to kick someone out.

If we have 3 frames available:



This produces a total of 9 page faults. Let's add a frame.

4 frames:



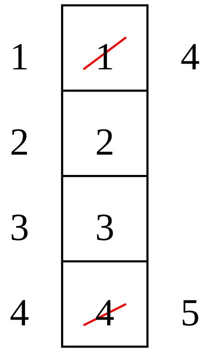
We got 10 page faults! This is very counterintuituve. We would expect that adding available memory would make page faults occur less frequently, not more. Usually, that is the case, but we have here an example of *Belady's Anomaly*. Here, adding an extra frame caused more page faults.

FIFO replacement is nice and simple – we know our victim immediately. However, it may produce a large number of page faults given an unfortunate reference string.

Optimal Algorithm (OPT)

We can't do better than replacing the page that will not be used for longest period of time in the future, so let's consider that.

4 frames example:



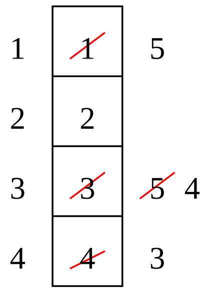
6 page faults.

Problem: we can't predict the future, at least not very accurately.

So it's not practical, but it is useful to see the best we can do as a baseline for comparison of other algorithms.

Least Recently Used (LRU) Algorithm

Select the frame that we have not used for the longest time in the past.



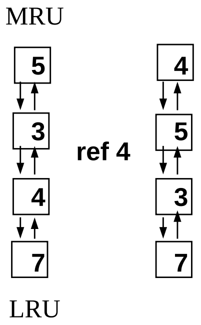
8 page faults. So we did better than FIFO here, but it's not optimal.

This one can actually be used in practice.

It can be implemented directly by maintaining a timestamp that gets updated each time a frame is accessed. Then when a replacement decision is needed, the one with the the oldest time stamp is selected.

This is expensive in terms of having to update these timestamps on every reference, and involves a linear search when a victim is being selected.

A stack-like implementation can eliminate the search. When a page is referenced, it moves to the top of the stack. A doubly-linked list can be used to make this possible:



This requires 6 pointer updates each time a page is referenced that was not the most recently used. This is still expensive, and would require hardware support to be used in practice.

LRU Approximation Algorithms

LRU is a desirable algorithm to use, but it is expensive to implement directly. It is often approximated.

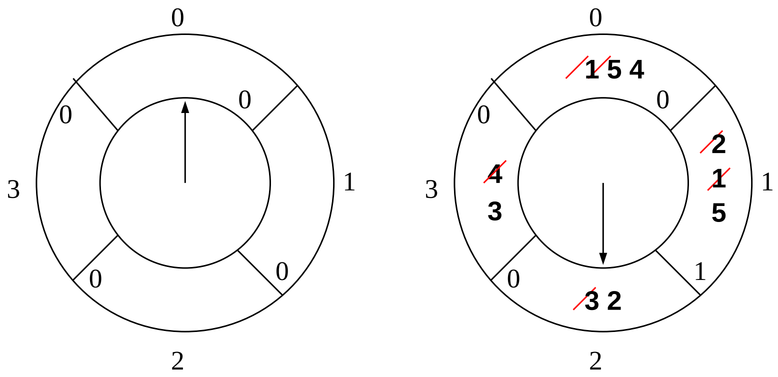
If we have one reference bit available, we can do a *second-chance* or *clock* replacement algorithm.

Here, we treat our collection of frames as a circular list. Each has a reference bit, initially set to 0. It is set to 1 any time the page is referenced. There is also a pointer into this list (the hand of a clock, hence the name) that points to the next candidate frame for a page replacement.

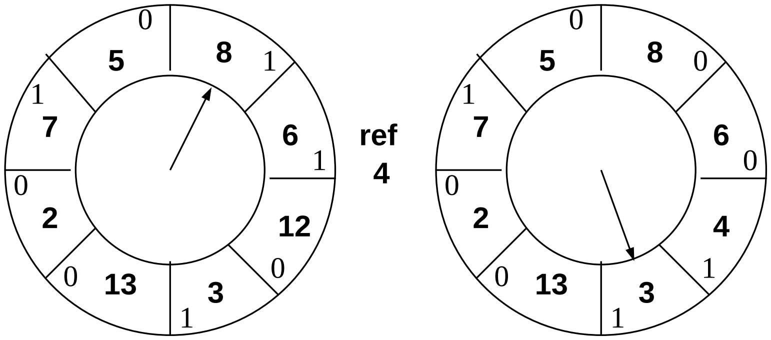
When a page replacement is needed, the frame pointed to is considered as a candidate. If its reference bit is 0, it is selected. If its bit is 1, the bit is set to 0, and the pointer is incremented to examine the next frame. This continues until a frame is found with a reference bit of 0. This may require that all frames be examined, bringing us around to the one we started with. In practice, this is unlikely.

It is called a second chance algorithm, as after a frame has had its reference bit set to 0, it has a second chance – if it is referenced before the pointer makes its way back around again, it will not be removed. Thus, frequently-used pages are unlikely to become victims.

Second-chance/clock does not do well with our running example.



10 page faults! We ran into the same unfortunate victim selection that we saw with FIFO with 4 frames. We need a larger example to see the benefit.



Pages 8 and 6 get their second chance, and may be referenced before the clock makes its way around again.

An enhancement to the clock algorithm called *Gold's Clock Algorithm* uses a second bit – a "dirty" bit in addition to the reference bit. The dirty bit keeps track of whether the page had been modified. Pages that had been modified are more expensive to swap out than ones that have not, so this one gives modified pages a better chance to stick around. This is a worthy goal, though it does add some complexity.

Counting Algorithms

One more group of algorithms to consider are those that keep track of the number of references that have been made to each page.

- *Least-frequently used (LFU)*: replaces page with smallest count. We haven't used it much, so maybe that means we never will.
- *Most-frequently used (MFU)*: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

It is not clear which approach is better. LFU would need to be combined with some sort of aging to make sure a page that is used a lot early on, getting a big count, can still leave eventually when its usage stops.

Implementations of these suffer from the same problems that the direct LRU approaches do. They're either too expensive because of needed hardware support or too expensive because of the searching needed to determine the MFU or LFU page.

Allocation of Frames

So far, we have considered how to assign pages of a process' logical memory to a fixed set of allocated frames (*local replacement*). We also need to decide how many frames to allocate to each process. It is also possible to select the victim from a frame currently allocated to another process (*global replacement*).

Each process needs a certain minimum number of pages.

- pages for instructions
- pages for local data
- pages for global data

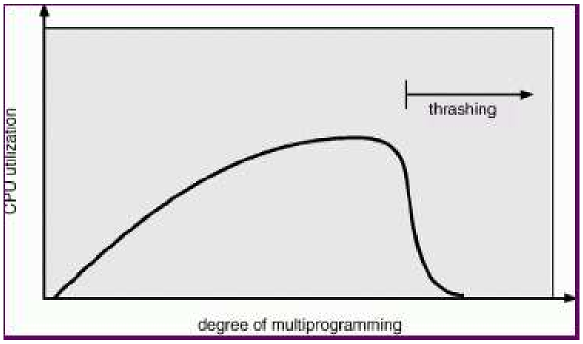
Allocation may be fixed:

- equal allocation – p processes share m frames, $\frac{m}{p}$ each
- proportional allocation – processes that have more logical memory get more frames
- priority allocation – high priority processes get more frames

Thrashing

Thrashing occurs when a process does not have "enough" frames allocated to store the pages it uses repeatedly. In this situation, the page fault rate will be very high.

Since thrashing leads to low CPU utilization (all processes are in I/O wait frequently having page faults serviced), the operating system's medium or long term scheduler may step in, detect this and increase the degree of multiprogramming.



With another process in the system, there are fewer frames to go around, and the problem most likely just gets worse.

Paging works because of locality of reference – a process access clusters of memory for a while before moving on to the next cluster. For reasonable performance, a process needs enough frames to store its current locality.

Thrashing occurs when the combined localities of all processes exceed the capacity of memory.

Working-Set Model

We want to determine the size of a process' locality to determine how many frames it should be allocated. To do this, we compute the set of logical pages the process has referenced "recently". This is called the *working set* of the process.

We define "recently" as a given number of page references in the process' history. This number is called the *working set window* is usually denoted Δ.

The working set will vary over time.

The appropriate selection of Δ allows us to approximate the working set size, WSS_i, of process P_i. If Δ is too small, WSS_i will not encompass the entire locality. If Δ is too large, pages will remain in the locality too long. As Δ increases, WSS_i grows to encompass the entire program.

The total demand for frames in the system is $D = \Sigma WSS_i$.

Thrashing by at least one process is likely when $D > m$, where m is the number of frames. If the system detects that $D > m$, then it may be appropriate to suspend one or more processes.

For the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, with $\Delta = 4$, the working set grows to {1,2,3,4} by time 4, but note that at time 8, it contains {1,2,5}.

Tracking the working set exactly for a given Δ is expensive and requires hardware support to hold the time stamps of the most recent access to each page. See the text for details of how to do this.

Windows NT and Solaris both use variations on the working set model.

Program Structure

We have stated that programs exhibit locality of reference, even if the programmer made no real effort to encourage it. But program structure can affect performance of a paging scheme (as well as the effectiveness of other levels of the memory hierarchy).

Consider a program that operates on a two-dimensional array:

```
int A[][] = new int[1024][1024]
```

and the system has a page size of 4KB. This means 1024 `ints` fit in one page, or one page for each row of our matrix A.

Program 1:

```
for (j=0; j<1024; j++)
  for (i=0; i<1024; i++)
    A[i][j] = 0;
```

Program 2:

```
for (i=0; i<1024; i++)
  for (j=0; j<1024; j++)
    A[i][j] = 0;
```

Assume a fixed allocation of 10 frames for this process. Program 1 generates 1024 × 1024 page faults, while Program 2 only generates 1024.

Segmentation

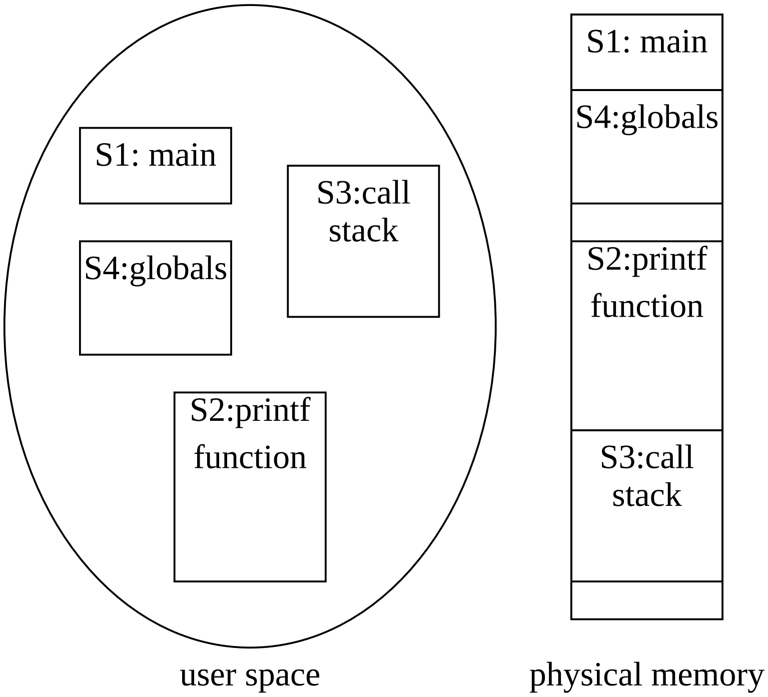
Another possible memory management scheme, sort of a hybrid of contiguous allocation and paging, is called *segmentation*.

Memory is allocated for a process as a collection of *segments*. These segments correspond to logical units of memory in use by a process:

- main program
- procedure, function, method
- object, local variables
- global variables
- common block (Fortran)
- stack

- heap space

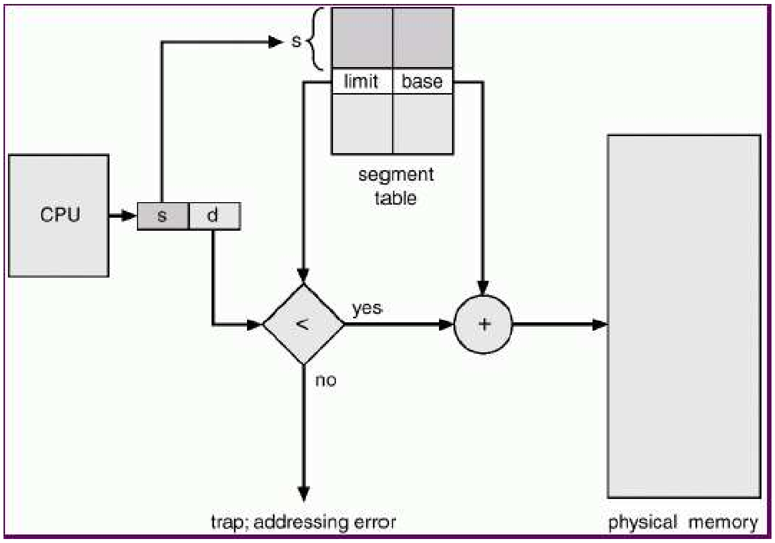
This can be thought of as a finer grained approach to contiguous allocation – smaller contiguous chunks, corresponding to these logical units – are scattered throughout memory.



With segmentation, a logical address consists of an ordered pair: (segment-number, offset)

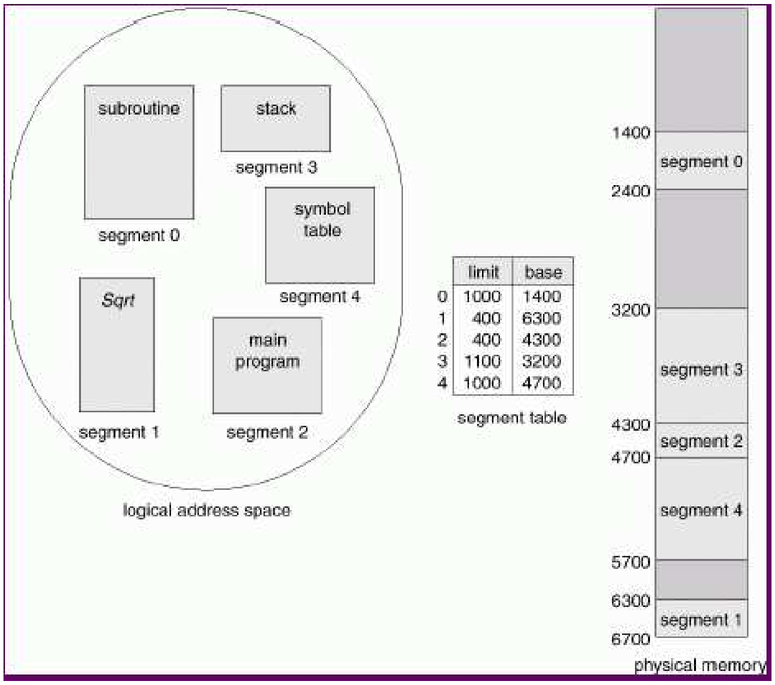
A segment table contains two entries for each segment:

- *base* – the starting physical address where the segment resides in memory
- *limit* – the length of the segment



Two registers locate the segment table in memory:

- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program; segment number s is legal if s < STLR.



With segmentation, segments may be relocated by moving the segment and updating the segment table. The segment number, and hence the logical addresses, remain the same.

Segment sharing is straightforward, as long as each process use the same segment number. This is required because the code in a segment uses addresses in the (segment-number, offset) format.

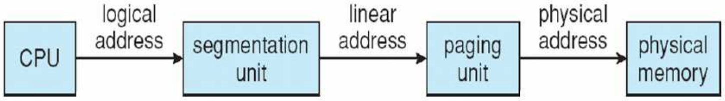
Allocation for the segments uses the same techniques as contiguous allocation (first-fit, best-fit), but since the chunks are smaller, there is less likelihood of some of the problems arising, though external fragmentation is there.

Intel Pentium Memory Management

Both paging and segmentation are actually used. In fact, many systems combine them. See the text for how they are combined in the Intel Pentium architecture.

The Pentium supports up to 16K segments, each with up to 2^{32} bytes of virtual address space.

The CPU generates a logical address, which is given to the *segmentation unit*. This produces a *linear address*. The linear address is given to the *paging unit*, which (finally) generates the physical address in main memory.



An OS running on a Pentium can choose to use only segmentation, only paging, or both.

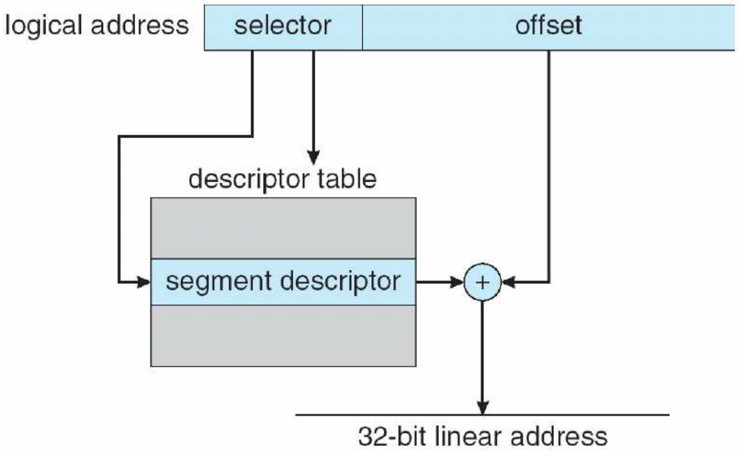
Many modern systems use only the paging model, where each process can have a single segment of 2^{32} bytes.

Segment descriptors are stored in two tables, each of which can hold up to 8K segments:

- *Local Descriptor Table (LDT)* – one per process, private to the process, describes each program's code, data, stack, etc.
- *Global Descriptor Table (GDT)* – shared by all programs, describes system segments, including those for the OS

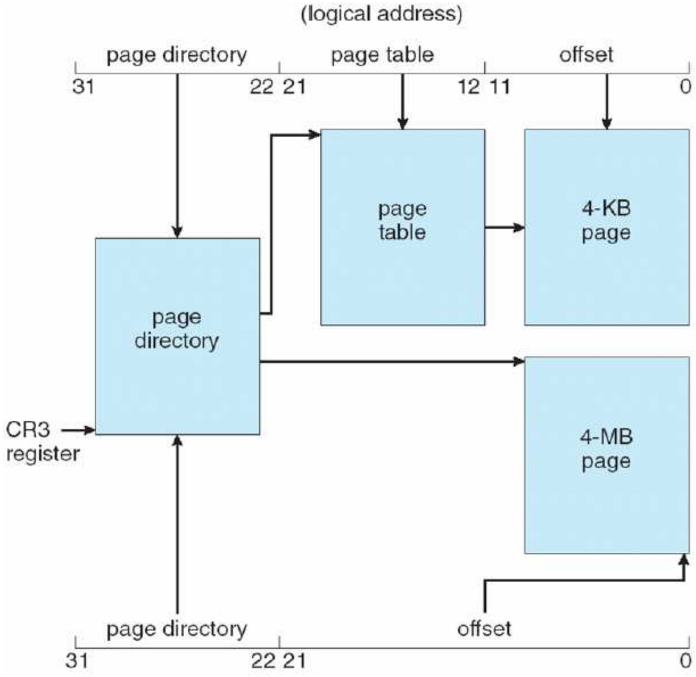
To access a segment:

- load a segment selector into a segment register
  - system has 6 segment registers, most important are `CS` – code segment, and `DS` – data segment
  - each selector is 16 bits: 13 bits form an index, 1 bit to select GDT(0) or LDT(1), 2 bits for a privilege level (0-3)
- this refers to an entry in the GDT ot LDT – which is itself an 8-byte value:
  - 32 bits base address, 20 bits for limit, 1 bit to determine 16- or 32-bit segment



The linear address is then converted by the paging unit into a physical address.

Page sizes can be either 4 KB or 4 MB. With a 4 KB page size, a 2-level page table is used. With a 4 MB page size, a straightforward page table is used.



Much more detail is available in chapter 3 of *Intel 64 and IA-32 Architecture Software Developer's Manual*, linked from the lecture page.

What's the Right Answer?

How might we compare strategies?

- how much hardware support is needed or available
- performance – more complexity = slower
- fragmentation?
- relocation – do we need dynamic binding?
- swapping – does it have to go back to the same place? Swap the whole thing?
- sharing segments/pages
- protection – need to ensure safe access