50004 - Operating Systems - Lecture 9

Oliver Killane

19/11/21

Lecture Recording

Lecture recording is available here

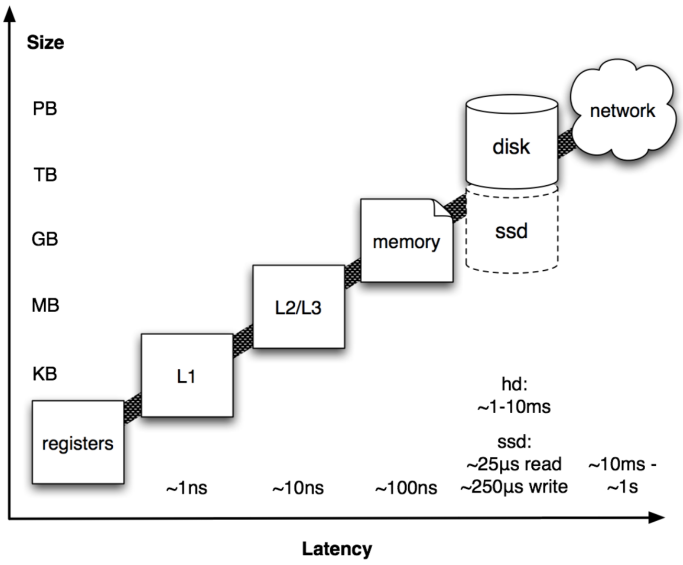
Memory Management

- **Von Neumann Architecture**
  Data and instructions are stored in the same memory (as opposed to Havard Architecture), every instruction requires a memory access.
- **Memory Allocation**
  The OS must be able to allocate memory to programs (i.e malloc/calloc/realloc).

  This needs to be fast & efficient as programs will allocate data on the heap often, and rerquire memory allocated to load.
- **Memory Protection**
  OS must prevent programs for certain memory. This ensures that processes can run in isolation as another process cannot corrupt their memory and cause a crash, likewise with the kernel and use processes.

  This is also important for security as processes may want to hide data (e.g passwords, keys etc) from other untrusted processes.

Memory Hierarchy



Note that the cost per size of memory increases from $L1 \rightarrow disk$

- **Registers**
  Accessible in a single clock, can store a limited size (usually 64 bits at most) and are very expensive.

  For example **x86_64** architecture uses fully-general purpose 16 registers

- **Cache**
  L3 is typically shared between cores. L2 & L1 per core.

  Caches can have differing levels of associativity, and hold values of recently accessed memory addresses.

Basic Concepts

- Memory Allocation
- Swapping
- Paging (Virtual Memory)
- Page Replacement Algorithms
- Working Set Model

Logical and Physical Addresses

Memory management binds the logical address space to a physical address.

- **Logical Address** Generated by the CPU and the address used by processes.
- **Physical Address** Address used by memory unit, refering to a location in physical system memory.

The use of logical addresses can change based on the **address-binding scheme**.

Address Binding Scheme

Determines when and how logical addresses are bound to physcial addresses, there are 3 such schemes:

- **Compile Time**
  When the program is compiled, the address binding is set.

  The compiler interacts with the OS's memory management.

- **Load Time**
  When a program is loaded into memory, the addresses are bound.

  Binding is done by the OS memory manager (e.g application loader).

- **Execution Time/Dynamic Address Binding**
  Memory binding postponed until execution starts, with binding done by the processor.

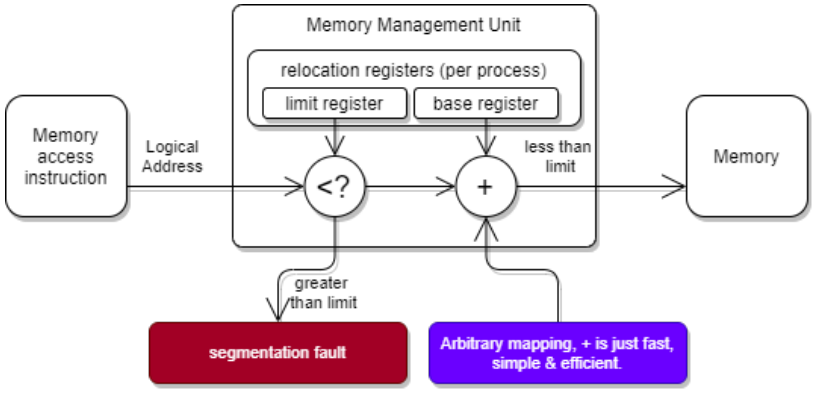
  This is the most common type used by practically all modern operating systems.

In Compile & Load time address-binding, the addresses used by the program are physical addresses. (Program binary is updated to contain these).

In Execution time binding logical addresses are used, and the CPU translates these to physical addresses at runtime.

Memory Management Unit

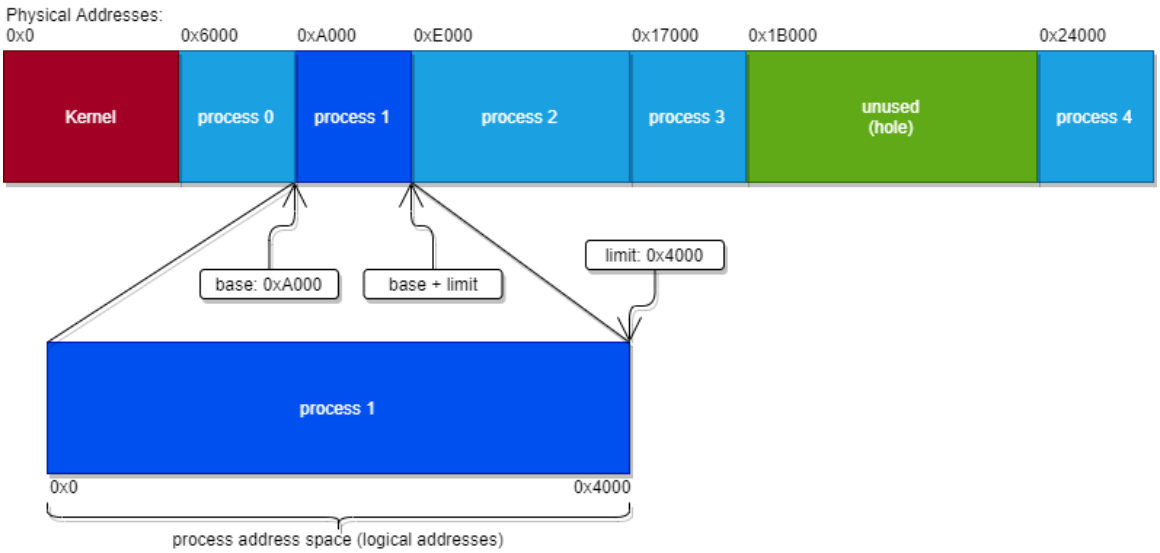
A hardware device to map logical to physical addresses.



The **base register** holds the smallest physical address, the **limit register** holds the highest logical address. This way the **MMU** can ensure:

$$base \leq \text{translated address} < base + limit$$

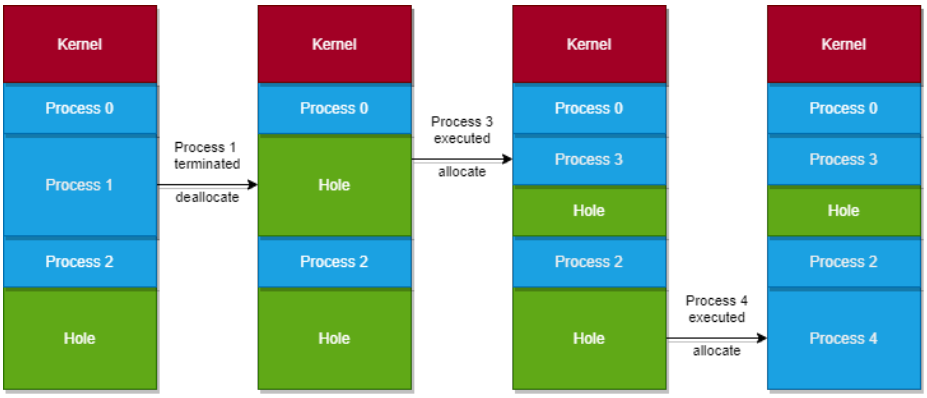
By ensuring processes can only access certain contiguous sections of memory, memory of the kernel and other processes is protected.



Multiple Partition Allocation

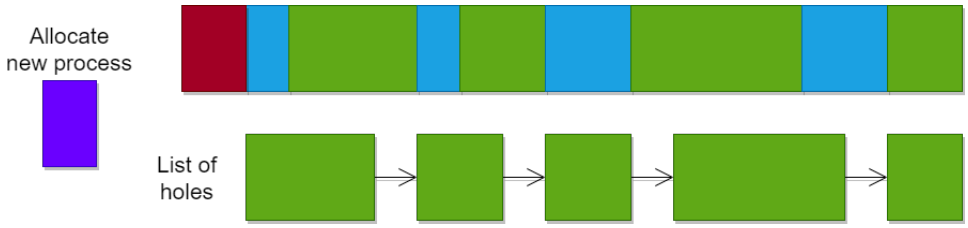
- **Hole** A contiguous section of unallocated memory.
- **Creation/Destruction**
  When a new process is started, the OS allocates the required memory from a sufficiently large hole.

  When a process is terminated, its memory is deallocated, and now free to be allocated to another process.
- **OS Requirements**
  The **OS** must keep track of which partitions have been allocated and which are free (holes).



Dynamic Storage Allocation

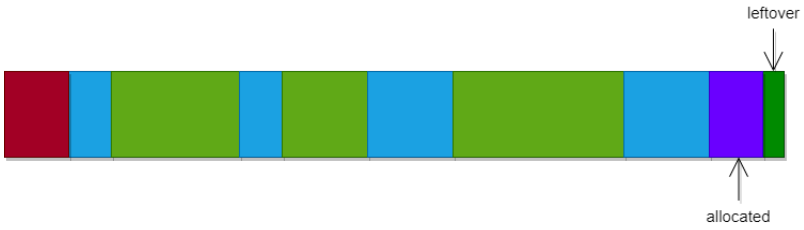
When allocating request for a certain size from the available list of holes and their sizers, held by the OS.



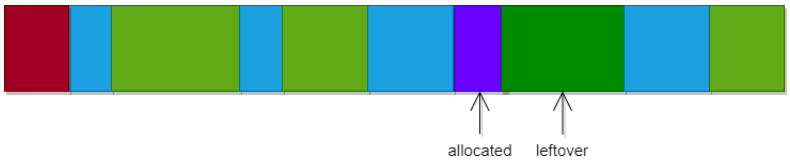
- **First Fit** Allocate in the first hole that is big enough. (Fast & Simple)

- **Best-Fit** Allocate the smallest hole that is large enough.

- Unless the list of holes is ordered, we must traver the whole list to decide which hole to use.
- Produces smallest leftover memory after allocation.



- **Worst-Fit** Allocate largest hole.
  - Search entire list (unless ordered)
  - The largest possible leftover produced.



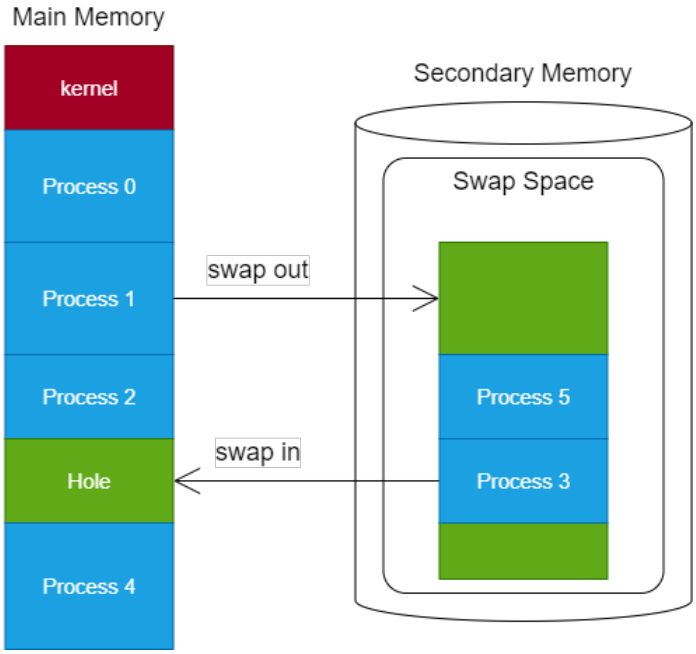
Fragmentation

- **External Fragmentation** Enough memory avaiulable, but not hole large enough. When we have a large number of very small holes. We can fix this by **compaction**:
  - Shuffle memory contents to place all free memory together in one block.
  - Can result in I/O bottlenecks, as requires a very large amount of copying.
- **Internal Fragmentation** Allocated memory larger than the requested. e.g Partition allocated, but program does not make use of all the partition. Wasted memory results in total memory available being low.

Swapping

The number of processes is limited by the amount of available main memory. (Note: only running processes need to be in main memory)

We can supplement main memory with **swap space** (can be a partition or file) on secondary storage (e.g HDD or SSD). However we must be aware than transfer times are long, and we must bring any process that is scheduled back into main memory.

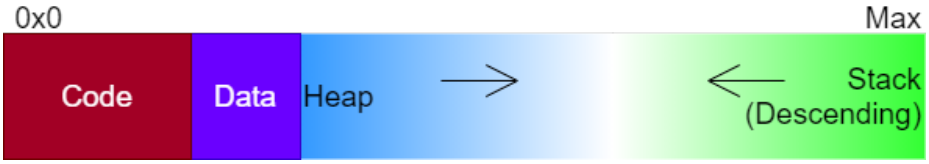


Virtual Memory with Paging

Virtual memory is an abstraction to separate logical memory from physical memory.

- Not all of an executing process needs to be in memory for execution.
- Logical address space can be much larger than the physical address space.
- Address spaces can be shared between several processes.
- Process creation can become more efficient.
- If page size is fixed, external fragmentation is impossible.

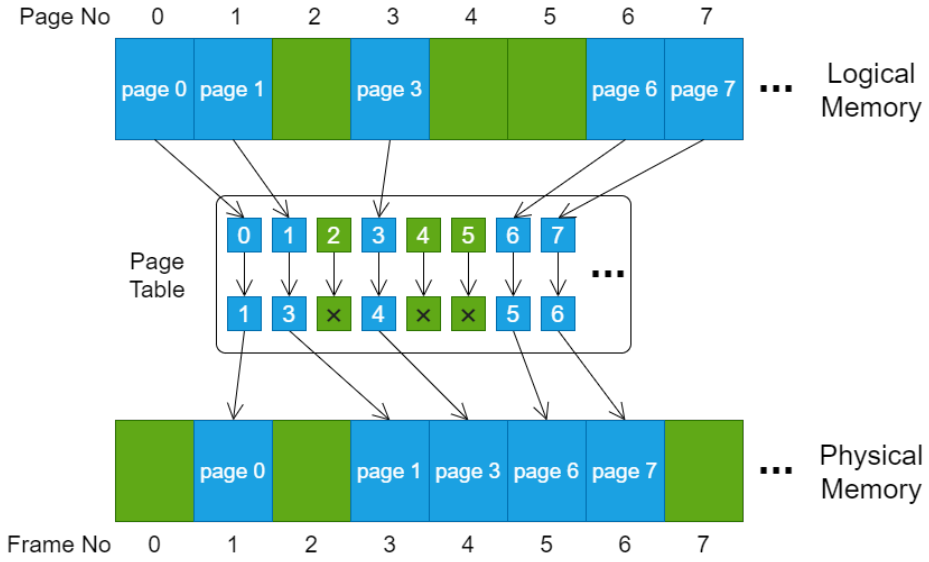
Virtual Address Space



- **Frames**
  Fixed-size blocks of physical memory. The **OS** must keep track of all free frames.
- **Pages**
  A block the same size as a frame, in logical memory (effectively a logical frame).
- **Program Start**
  For a program of size n frames.
  1. Find n free frames, load the program into that memory.

2. Create a page table for the process, mapping logical pages to physical frames.

- **Program Termination** Destroy the page table, freeing all associated frames.



The advantages of this approach are:

- Less internal fragmentation (If a page is not yet used, it is not allocated in memory).
- Less external fragmentation (When allocating alot of memory, split into pages and distribute across physical memory).
- Contiguous logical addresses can be physically non-contiguous.

Page Size

Small Page Size

- Less internal fragmentation.
- Potentially less memory to swap.
- Larger Page table.

Best for efficient memory usage.

Large Page Size

- More internal fragmentation
- Smaller Page table.

Best for low address lookup overhead.

Context Switch

When switching process, the OS must locate the page table of the new process. Set the base register to point to the table in memory, and clear the invalidated cache from the TLB.

TLB

Translation Lookaside Buffer is used to cache the mappings of **Virtual** to **Physical** addresses.

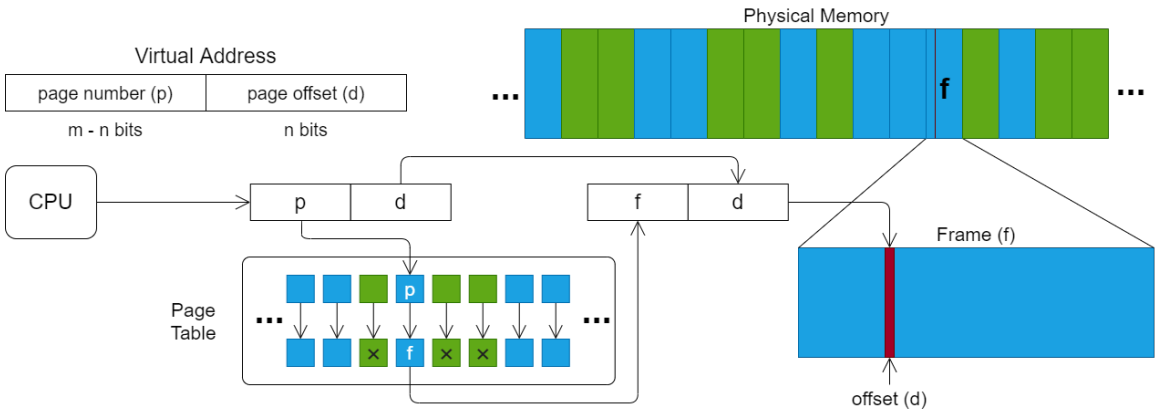
When a virtual address is used & its corresponding physical address found, it is added to the cache so that if the virtual address is used again the physical address does not need to be recomputed.

Address Translation

Each memory address is split into a **page number**(p) and a **page offset**(d)

- **Page Number**
  The index of the page, used to get the base-address of the corresponding frame from the page table.
- **Page Offet**
  The offset through the page/frame, which is combined with the base-address to get the physical address.

For logical address space 2^m with page size 2^n:



For example:
A 16-bit (big endian) system. 1KByte page size. Each page entry's LSB is the valid bit, and second LSB is modified (dorty) bit. Page table:

0	1	2	3	4
$0x2C00$	$0x0403$	$0xCC01$	$0x000$	$0x7C01$

Find the physical memory addresses of the following virtual addresses:

$$0xB85 \quad 0x1420 \quad 0x1000 \quad 0xC9A$$

Get addresses	0xB85	0x1420	0x1000	0xC9A
	↓	↓	↓	↓
Divide address by 1024 to get page #	2	5	4	3
	↓	↓	↓	↓
Get Page Table Entry	0xCC01	PAGE FAULT (past end)	0x7C01	0x000
	↓	↓	↓	↓
Check Valid & Dirty Bits	VALID	N/A	VALID	PAGE FAULT (invalid)
	↓	↓	↓	↓
Divide entry by 1024 to get frame	0x33	N/A	0x1F	
	↓	↓	↓	↓
Add frame onto front of offset	0xCF85	N/A	0x7C00	N/A

Memory Protection

We associate protection bits with each pages.

We associate a valid-invalid bit with each entry in the page table.

- **Valid** Associated page is in process' logical address space.
- **Invalid** Page is missing!
  - Page not in process' logical address space (page fault).
  - Must load page from swap space (on disk) (see demand paging).
  - Incorrect access of some kind.

Demand Paging

Demand paging is a method of memory management where pages in the swap space on the disk are only loaded into main memory when an access attempt is made.

This is opposed to **anticipatory paging** (the norm) where program's pages are loaded from the swap when it starts running, in anticipation of them being accessed.

Page Table Implementation

- **Page Table** is kept in main memory.
- The **Page-Table base register** (**PTBR**) points to the page table's start
- The **page-table length register** (**PTLR**) indicates size.

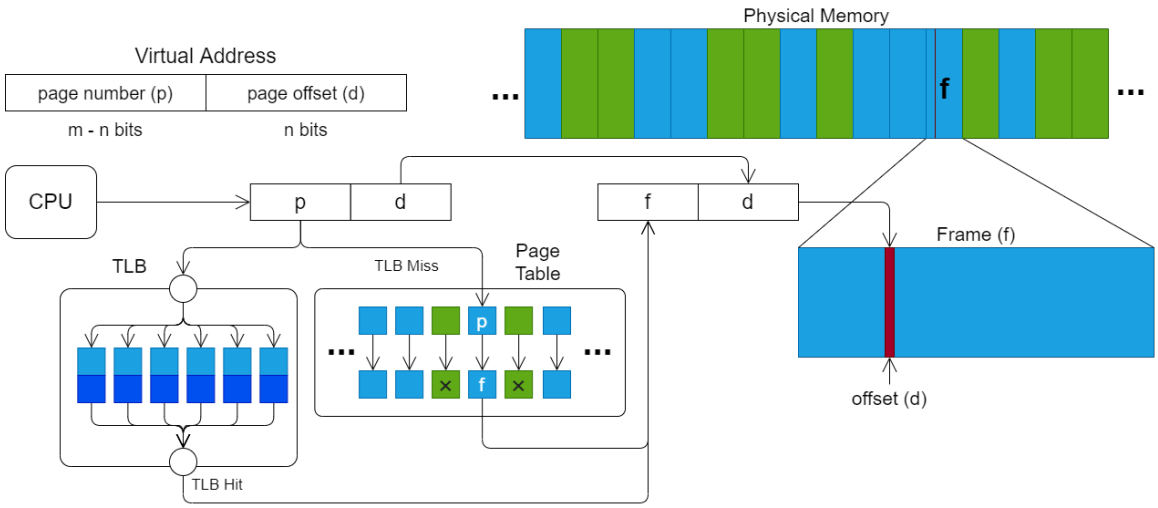
However using the page table for every memory access is slow (requires access to page table in memory), so caching is used.

Note that for kernel mode operations, a bit is set in the page table

Associative Memory

Also called **CAM** (Content addressable memory), it is a special type of memory optimised for search (making use of parallelism).

Generally much more expensive than **RAM** and can only be used for a fixed memory allocation format (cannot search for different data types after creating the silicon).



Translation Look-aside Buffer

An **associative memory** cache that stores translations of virtual pages to physical frames.

Address-Space IDs

Some **TLB**s store **address-space ids** (**ASID**s) in entires. As a result the TLB does not have to be fully wiped when context switching, as it can recognise that old entries are invalid using the associated **ASID**.

This potentially improves performance as entries may survive a short context switch, and no cycles are wasted in removal.

The **MIPS** and **ARM** architecture's TLB uses this approach with 8-Bits ASIDs, and many other architectures have added this feature.

Kernel Page access in System Calls

When a process makes a system call, and the system call handler runs in privileged mode int that process, it may need to access kernel pages.

To ensure safe access, the page table will have a **supervisor** bit. If set for a translation, it means that when in kernel mode, the page can be accessed.

For example with page table:

Page	*Frame*	*Valid*	*Supervisor*
0	3	1	0
1	5	1	0
2	14	1	1

A process cannot access page 2 without a segfault, however if in kernel mode following a system call, that page can be accessed.

An alternative implementation would use separate page tables for user space and kernel mode execution.

Effective Access Time

Given that:

- (ω) Memory cycle time.
- (ϵ) Associative Lookup time.
- (α) Hit Ratio for TLB or Associative Registers (generally more $\Rightarrow$ higher hit rate)

$$\text{Effective Access Time (EAT)} = (\epsilon + \omega)\alpha + (\epsilon + 2 \times \omega)(1 - \alpha)$$

(If hit, one memory lookup, else 2.)

Page Table Types

Break up the logical address space into multiple page tables:

Multi-Level/Hierarchical Page Table

Page table size can be given by:

$$\text{page table size} = \frac{\text{Address Space Size}}{\text{Page Size}} \times \text{Entry Size}$$

As a result for very large address spaces, with small page sizes, the page table can become large.

This is an issue as:

- If page table is larger than a frame - added complexity.
- Page table uses lots of memory, even if a process does not access any/many pages.

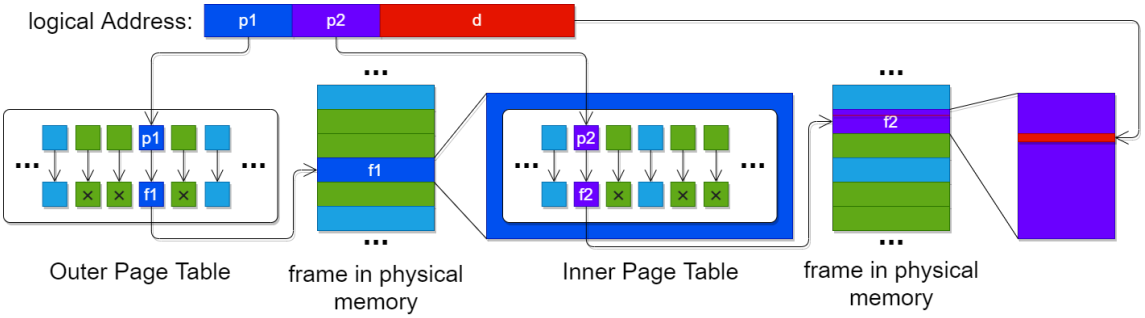
To resolve this we can use multi-level page tables - trees containing page tables.

For n levels the page number is partitioned into n sections. To access a page:

1. **Get Outermost Page table** Typically in a single frame.

2. **For each partition of the page number**
   (a) Get index from partition of page number.
   (b) Get entry at index in the current frame (a page table).
   (c) If entry is invalid, empty or dirty, then page fault.
   (d) Else get the frame from the entry. Set this as the current frame.
3. **Get physical address** In the current frame, add the page offset.

An example of a two-level page table is below.



The disadvantage of this system is there are more memory accesses to allow a process to access memory (in a TLB miss).

For example a system with memory access time of $100ns$ and TLB access of $20ns$.

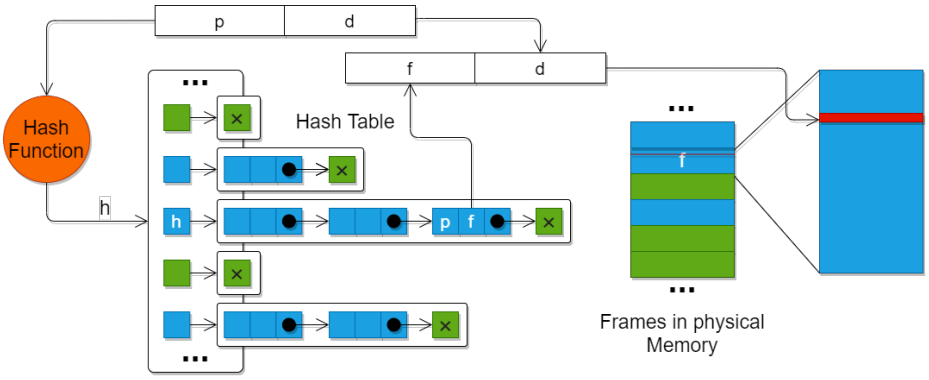
Hit Rate	Single Level	4-Level
80%	$0.8 \times (20 + 100) + 0.2 \times (20 + 2 \times 100) = 140ns$	$0.8 \times (20 + 100) + 0.2 \times (20 + 5 \times 100) = 200ns$
98%	$0.98 \times (20 + 100) + 0.02 \times (20 + 2 \times 100) = 122ns$	$0.98 \times (20 + 100) + 0.02 \times (20 + 5 \times 100) = 128ns$

As unpaged memory access requires only $100ns$ we can see the performance of paged memory access gets closer as hit rate increases, even at large paging levels.

Hashed Page Table

A hash table to map page numbers to frame numbers. (A normal page table is effectively a Hashed Page table with identity as the hash function.)

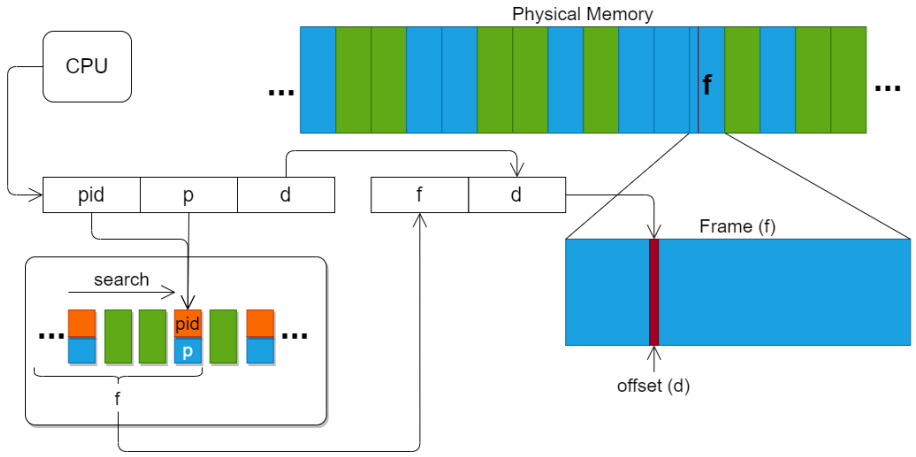
By using more complex hash functions, we can reduce the size of the page table, at the expense of conflicts. Where we can use a linked list to resolve conflicts.



Inverted Page Table

A page table containing an entry for every page frame of memory, containing:

- Virtual Address of page stored at location.
- Information on the owning process (e.g process ID).



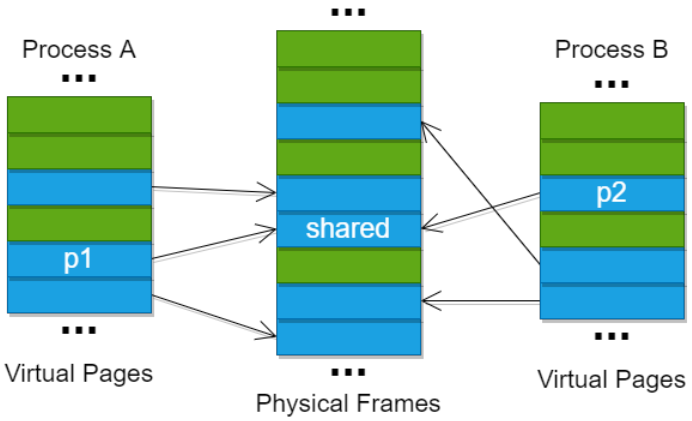
Compared with the standard page table implementation:

- Decreases memory needed to store page table.
- Increased time to search table (e.g must filter through enties with other process' IDs).
- Can use a hash table to limit linear search to a few entries.

Shared Memory

Processes can access shared memory by having pages in two processes point to the same frame.

Once the pages are setup to achieve this, the kernel does not need to be involved.



This is useful for IPC, as well as other uses such as sharing libraries. For example **System V**'s API (note more modern systems use **mmap**):

```c
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/shm.h>

/* To use the System V IPC, you need a key from an associated file */
key_t ftok (const char *pathname, int proj_id);

/* Get a memory  sahred memory segment with key, or create a new one.
 * There are several flags for creating private shared memory, and setting
 * permissions (use shmflag).
 */
int shmget (key_t key, size_t size, int shmflg);

/* Attach shared memory segment to the address space of a process. */
void *shmat (int shmid, const void *shmaddr, int shmflg);

/* Detach a share memory segment from the process. */
int shmdt (const void *shmaddr);

/* Issue a command to control the shared memory segment. */
int shmctl (int shmid, int cmd, struct shmid_ds *buf);

struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};

struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
```

```
40      gid_t           cgid;       /* Effective GID of creator */
41      unsigned short  mode;       /* Permissions + SHM_DEST and
42                                     SHM_LOCKED flags */
43      unsigned short  __seq;      /* Sequence number */
44  };
```

System V

One of the first commercial **Unix** Operating systems originally developed by **AT&T** and released in 1983.

Was a competitor/rival to **BSD** with much cross pollination fo features occurring between the two. It also had several distributions by other companies such as Oracle's Solaris OS.