Contents of Lecture 5 on Memory Management

- Dynamic memory allocation
- Linux kernel memory allocation
- Introducing virtual memory
- Assigning virtual addresses to symbols — how link-editors work

Dynamic Memory Allocation

- The C library functions `malloc` and `free` are used for **dynamic memory allocation**.
- So, `malloc` is called to request `size` bytes of memory and returns either
  - a pointer `ptr` to the allocated memory, or
  - NULL, if the request could not be performed.
- The memory area is returned by calling `free(ptr)`.
- While these things probably are well-known, it's important to note that the problem becomes interesting because:
  1. the allocations can request different sizes
  2. calls to `malloc` and `free` can come in any order
- How could we implement malloc/free without these constraints?

Dynamic Memory Allocation in TAOCP

- TAOCP = The Art of Computer Programming
- Volume 1, Section 2.5 of Knuth's work discusses dynamic memory allocation.
- We will discuss two techniques for implementing malloc/free:
  - Linked-lists
  - Buddy system

List Approach to Dynamic Memory Allocation

- Assume we have a large pool of memory of size N bytes (e.g. 100KB).
- At a `malloc(n)` call, we request n bytes.
- At a `free(pointer)` call, we return a pointer to n bytes. To let free figure out that we are returning n bytes, we may store n at the address `pointer - sizeof(size_t)`.
- After a number of malloc and free calls, the memory pool will consist of a number of **reserved** and **available** blocks.
- In each available block is stored the size and a pointer to the next available block.

Example in C

```
#include <stddef.h>

#define N (8 * 1024*1024 / sizeof(size_t))

typedef struct list_t list_t;
struct list_t {
        size_t  size;   /* size including list_t */
        list_t* next;   /* next available block. */
        char    data[]; /* C99 flexible array. */
};

static size_t   pool[N] = { N * sizeof pool[0] };
static list_t   avail   = { .next = (list_t*)pool };
```

- The first bytes in `pool` are interpreted as a list struct.
- Using `size_t` for the pool elements, we can align and initialize it easier.

C Code for Allocation using First Fit

- TAOCP Algorithm A: First-fit method.
- Requested size is `size`.
- We start with incrementing `size` by the size of a `list_t`.
- Use two pointers, `p = &avail` and `q = p->next`
- Search the list until a sufficiently large block is found or end of list.
- Issues:
  - What should we do if we find a block with `q->size == size`?
  - What should we do if instead `q->size` is larger than `size`?

C Code for Deallocation

- The list is sorted by address, i.e. "relative position" in `pool`.
- First we need to find the list from which our data comes.
- Find the list struct from the returned address by subtracting the size of a list.
- Why should you not do as follows? — `data` is a void pointer.

```
list*   r = (list_t*)data - sizeof(list_t);
```

- or

```
list*   r = data - sizeof(list_t);
```

- but rather:

```
list*   r = (char*)data - sizeof(list_t);
```

Inserting `r` into the List

- Use two pointers, `p = &avail` and `q = p->next`.
- Search the list while `q != NULL` and `r < q`.
- Check whether the block pointed to by `r` can be merged with either that pointed to by `p` or `q`, or both.

Alignment

- The code you saw does not take alignment into account.
- A requested size larger than A should be aligned on A if A is not greater than the maximum alignment requirement.
- The maximum alignment requirement might be 8 for `double` and 16 for long double or a SIMD vector register.
- You must assure that the memory pool itself is aligned.

Increasing the Pool Size

- UNIX has a system call `brk` which sets the address of the end of the heap.
- There is a suitable library function `sbrk` which is preferable to use.
- With `sbrk(size)` we request `size` of additional heap space and then `sbrk` will call `brk`.
- Areas allocated with different calls to `sbrk` can be merged but `malloc` must be aware of that other functions also may call `sbrk`.

Other Problems and Solutions

- **Problem:** If only a few words remain in a block, we will create a very small block which will probably be useless but take up time to skip the during search.
  Solution: Allocate these words too. We now need to save the real size of the block for deallocation (`void free(void*)` needs that anyway).
- **Problem:** The first few blocks tend to become split into very small pieces which always must be skipped when searching for a larger block.
  Solution: See the *Next-fit* method on the next page.
- **Problem:** Deallocating blocks requires us to search the list to determine whether we can merge with the previous block.
  Solution: Use a doubly linked list.

Variants

- *Next-fit* remembers the where the search stopped last time and starts the next search there.
- *Best-fit* searches the entire list to find the smallest block which is sufficiently large (may be too time-consuming).
- *Quick-fit* uses an array of lists for blocks with known popular sizes — typically adapted to your application.

Buddy System Illustration

Operation	Memory pool								# Available blocks
	0	64	128	192	256	320	384	448	
Start	512								1
Allocate 70	256				256				2
	A		128		256				2
Allocate 35	A		B	64	256				2
Allocate 80	A		B	64	C		128		2
Deallocate A	128		B	64	C		128		3
Allocate 60	128		B	D	C		128		2
Deallocate B	128		64	D	C		128		3
Deallocate D	256				C		128		2
Deallocate C	512								1

Buddy System Data Structure

- memory: array of 2^N bytes assumed to start at address 0.
  You must compensate for this by subtracting the real start from a block address.
- *freelist*$[K]$ refers to the freelist with available blocks of size 2^K
- Each block has the following fields:

```
unsigned    reserved:1; /* one if reserved. */
char        kval;       /* current value of K. */
list_t*     succ;       /* successor block in list. */
list_t*     pred;       /* predecessor block in list. */
```

Buddy System Allocation

- Increase size to be the smallest power of two: $size = 2^K$
- Find the first list J with an available block; $J \geq K$
- If no such J exists, then return `NULL`.
- For each $I : K < I < J$ split the block in two pieces, set their `kval`, `reserved`, `succ` and `pred` attributes, and put them into $freelist[I - 1]$
- Return the first block in $freelist[K]$.

Deallocation

- Increase size to $size = 2^K$
- Then find address of buddy:

```
buddy = pointer ^ (1 << K); // if pool at address 0...
```

- Correct:

```
buddy = start + ((pointer - start) ^ (1 << K));
```

- Check if `buddy` is free and has the same `kval`, if so
  - merge the two blocks, i.e. update pointers and `kval`
  - call deallocate recursively with the merged block and doubled size.

Explanation of Deallocation

- Assume the pool starts at address 0 and that both pointer and buddy point to a block of size 2^K.
- We have the address to one block, `pointer`, and want to find its buddy.
- We either have | pointer | buddy | or | buddy | pointer |
- The addresses to these are | xxxxx | 0 | K zeros | and | xxxxx | 1 | K zeros |, but we don't know which one is which and we don't want to test using compare and branches.
- 2^K = `1 << K`, i.e. | 000001 | K zeros |
- pointer XOR 2^K == buddy

Analysis of the Buddy System

- Both allocation and deallocation are fast.
- Can create large fragmentation, e.g. allocate(513) will reserve 1024 bytes. (actually, we must also account for the sizes of each block's attributes too).
- Inefficient with alternating allocate/deallocate: first lots of splits and then lots of merges.
- Modifierad version is used in the UNIX System V Release 4 kernel which delays the merging in the hope that the small blocks will be useful.

Fragmentation

- The linked-list methods may create useless available blocks.
- This is called **external fragmentation**.
- The buddy system on the other hand may allocate too much for each request. This also happens with large pages in a virtual memory system, and is called **internal fragmentation**.

The Linux Kernel Slab Allocator

- As you will soon learn, the memory is divided into pages of e.g. 4 KB.
- The worst problem with the buddy allocator is that it easily causes internal fragmentation: if you need 33 pages, you must request 64 pages
- The **Slab allocator** allocates memory using the buddy allocator
- Each slab is used for a certain type of object (with the same size)
- When an object is needed from a slab, it is removed
- When all objects taken from a slab are deallocated, the slab is returned (and the memory can be used e.g. for some other object type)

Multiprogramming without Virtual Memory

- Using two extra hardware registers for **base** and **limit**, we can have multiple processes in RAM at the same time and these processes cannot write in each others' memory.
- So we have achieved support for:
  - Multiprogramming — which enables switching processes while waiting for I/O.
  - Protection — fewer insane problems.
- The main problem left to solve is **dynamic memory allocation**.
- The main restriction left possible maximum process size — big problem.

We want to have a better memory system than this!

What is Virtual Memory?

- Primary memory (RAM) is smaller and faster than secondary memory (hard disks) so if the RAM is too small we can copy things to the disk.
- Alternative view: RAM is a cache for the disk.
- Without virtual memory, unless RAM is sufficiently large, the programmer must use both memories and write code to transfer code and data between them.
- There usually are system libraries to help with this (mess) and the feature is called managing overlays.
- You reserve some memory areas and make sure different parts of the program loaded from disk are not needed simultaneously.
- It has actually came back: the SPU's of the Cell processor have only 256 KB RAM which can be "extended" with overlays.
- Virtual memory is a technique to do this automatically and was invented in 1956.

History of Virtual Memory

- Invented in 1956 by Fritz-Rudolf Güntsch for his PhD thesis.
- First implemented in the Atlas computer 1959-1962 at U. of Manchester in England.
- First commerical machine with VM was the Burroughs B5000 from 1961 from the US.

Symbol Addresses in Programs

- With the invention of virtual memory, programmers could think of only one level of memory instead of both.
- Much more convenient.
- Essentially, RAM is divided into blocks of 4 KB or 8 KB pages.
- The addresses produced by the CPU are virtual addresses.
- The key issue is: where is the data of a virtual address located?
- It's obviously either on disk or in RAM — or both.
- Every address can be split into a page number and an offset within that page.
- Assume the page size is `SIZE`.

```
page   = addr / SIZE;
offset = addr % SIZE;
```

Mapping from a Virtual to a Physical Page Number

- A huge array indexed by virtual page numbers can be used to map to a physical page number.
- This array is called the **page table**.
- It contains information about each virtual page of a process.

```
typedef struct {
        unsigned int    page:26;        /* Swap or RAM page. */
        unsigned int    inmemory:1;     /* Page is in memory. */
        unsigned int    ondisk:1;       /* Page is on disk. */
        unsigned int    modified:1;     /* Page was modified while in memory. */
        unsigned int    referenced:1;   /* Page was referenced recently. */
        unsigned int    readonly:1;     /* Error if written to. */
} page_table_entry_t;
```

- For every memory access, this table can be used to find whether the data is in memory.
- If it's not in memory, it must first be fetched from disk.
- Actually, it might not be in the disk either — in which cases??

Accessing the Page Table

- With the page table, the address from the CPU has its virtual page number replaced with the physical page number stored in the page table:

```
static void translate(unsigned virt_addr, unsigned* phys_addr, bool write)
{
        unsigned        virt_page;
        unsigned        offset;

        virt_page = virt_addr / PAGESIZE;
        offset = virt_addr & (PAGESIZE - 1);

        if (!page_table[virt_page].inmemory)
                pagefault(virt_page);

        page_table[virt_page].referenced = 1;

        if (write)
                page_table[virt_page].modified = 1;

        *phys_addr = page_table[virt_page].page * PAGESIZE + offset;
}
```

- Hardware must do this translation otherwise too slow.
- It's still too slow to read the page table in RAM every access.
- How would you solve this problem?

The TLB

- A special hardware table called the **translation lookaside buffer** is used to remember recent virtual to physical page translations.
- The TLB might have 64 or 128 entries.
- All entries are checked concurrently to increase speed.
- There is one TLB for instructions and one for data.
- Thus the CPU sends in the virtual page number and either receives a physical page number or a hardware **exception** is triggered.
- The exception is called a TLB fault.
- The kernel then looks up the virtual page in the page table, store the translation in the TLB and re-executes the faulted instruction.
- Some machines, including Power, do the TLB miss handling in hardware for increased speed (MIPS does it in the kernel).

Page Faults

- Suppose a TLB page fault has happened and the data is not in RAM.
- Then the kernel must decide on a page in RAM to overwrite and read the needed data from the disk.
- How can we inform the page table entry of the previous owner that it's page has disappeared (or been moved to the disk to be correct)?

Example TLB Attributes

- valid bit for the translation
- virtual page number
- physical page number
- process identifier: not UNIX PID — MIPS TLB use a 5-bit id
- write protection
- modified bit
- referenced bit to help page replacement algorithms
- size of translated area: e.g. ARM11 uses two bits for
  - 4KB,
  - 64 KB,
  - 1 MB and
  - 16 MB.

Core Map

- Every physical page in RAM has a struct with which we can find the current owner:

```
typedef struct {
        page_table_entry_t*     owner;  /* Owner of this phys page. */
        unsigned                page;   /* Swap page of page if assigned. */
} coremap_entry_t;
```

- If the current data in that page was modified, it must first be written to disk.
- We will next go through some parts of a simple implementation of a virtual memory system and the next lecture we will look at problems with this implementation.
- This code is the basis for Lab 3.
- After this, we will learn how link-editors assign addresses to symbols.

Link-Editor Terminology

- **Compile time:** The time when one file is translated to object code (machine code or a Java class file).
- **Load time:** It is actually normally called **link-time**. The link-editor program `/usr/bin/ld` combines object modules to produce an executable program (for most languages e.g. C/C++).
- **Execution time:** While the program is running.
- **Static linking:** The link-editor creates an executable file with all symbols included. Results in larger executable files. Command: `gcc -static file.c`.
- **Dynamic linking:** The link-editor creates an executable file, but some symbols are missing and must be found during execution-time. Eg finding `printf` is postponed.

The ELF Format for Object and Executable Files

- ELF stands for **Executable and Linkage Format**.
- ELF describes the following file types on Linux and Solaris:
  - Relocatable files: file name extension `.o`, produced with `gcc -c file.c`. Such files are output from the assembler and are input to the link-editor.
  - Statically and dynamically linked executables.
- An ELF file consists of an ELF header, an array of program headers, an array of section headers, and the data corresponding to each header.
- In general, program headers are used by the kernel and the section headers are used by the link-editor.

ELF Sections

- Example sections: instructions, global variables, symbol table, relocation entries for instructions, relocataion entries for global variables.
- There are three instruction sections (`.init`, `.text` and `.fini`) but the program header refers to all three: the linker need to know the details but not the OS.
- Optional sections include debugging information — the most sophisticated format is DWARF3 which will become an IEEE standard. GDB implements most of this.
- One can also easily add new sections which can be useful when some extra information should be passed to the execution environment.

The ELF Header of a File

```
byte    e_ident[];      /* 0x7f 'E' 'L' 'F' */
short   e_type;         /* Eg relocatable. */
short   e_machine;      /* Eg PowerPC32. */
int     e_version;      /* ELF version. */
int     e_entry;        /* Address of first instruction. */
int     e_phoff;        /* Where is prog header array? */
int     e_shoff;        /* Where is section header array? */
int     e_flags;        /* Endianness, 32/64. */
short   e_ehsize;       /* Sizeof ELF header. */
short   e_phentsize;    /* Sizeof prog header. */
short   e_phnum;        /* Number of prog header. */
short   e_shentsize;    /* Sizeof section header. */
short   e_shnum;        /* Number of section header. */
short   e_shstrndx;     /* Section number for string table. */
```

The Program Header

```
int     p_type;         /* Read to process memory or not. */
int     p_offset;       /* Where in ELF file is data. */
int     p_vaddr;        /* Load at this address. */
int     p_paddr;        /* Load at this address (rarely used). */
int     p_filesz;       /* Length of data in the file. */
int     p_memsz;        /* Length of data in memory (see below). */
int     p_flags;        /* Eg read, write, execute permissions in memory. */
int     p_align;        /* Maybe must start on a new page. */
```

- Explanation to why `p_memsz` may be greater than `p_filesz`: Global variables with no explicit initialising expression are always initialised to zero. The difference between `p_memsz` and `p_filesz` tells the OS how many bytes should be initialised to zero at program startup. This technique reduces file sizes by avoiding to store zeroes in a file.

The Section Header

```
int     sh_name;        /* Offset into data of string table section. */
int     sh_type;        /* Eg progbits (code and data), symbol table. */
int     sh_flags;       /* Eg readonly or executable. */
int     sh_addr;        /* Virtual address when known. */
int     sh_offset;      /* Where in ELF file is data. */
int     sh_size;        /* Sizeof section data. */
int     sh_link;        /* Pointer to other relevant section. */
int     sh_info;        /* Pointer to other relevant section. */
int     sh_addralign;   /* Eg 8 if data contains a double. */
int     sh_entsize;     /* Eg 32 bytes for symbol table entries. */
```

- Relocation sections (should actually be called fill-in-the-now-known-address sections) use link and info to find the section to modify and the symbol table which now knows the addresses.

Link Editing

- Input is a set of object files, library names, and a list of directories which are used to look for libraries.
- The object files are checked for missing symbols which then are searched for in the libraries.
- The command `gcc main.c -L. -lmath` will search for missing symbols in the archive called `libmath.a` in the directory `.` (dot, i.e. the current directory).
- When all symbols have been found, we have a set of object files to "concatenate".
- Each object file will contribute with data and instructions to the final executable file (which has the default name `a.out` (also for dynamic linking)).

Creating the Executable File

- Conceptually all sections of the same type from all files are concatenated, and is put into the output file.
- So, first come instructions from file 1, then instructions from file 2, etc and then data from file 1, data from file 2, etc.
- Using this ordering, each section can be assigned an address.
- Each symbol is located at an offset from its section and can now be given a virtual address which is the address of its home section plus the offset.

UNIX Archive Files

- Libraries are stored in a different format: the UNIX archive which contains a set of relocatable files, and a symbol table describing its files.
- An archive is produced e.g. like this: `ar rcv libmath.a sin.o cos.o`.
- Common arguments to `ar` are:
  - `r` Replace previous file in the archive.
  - `c` Create the archive if it does not exist already.
  - `x` Extract a file from the archive: `ar x libmath.a tan.o` will try to copy a file `tan.o` from the archive to the current directory.
  - `v` Be verbose.

Page Table Implementation Strategies

A one-dimensional dense array is too large to store in RAM

1. Split the page table into two or three levels — costly at page faults.
2. Use a one-dimensional array but put it in virtual memory — needs a page table in RAM to map the other page tables.
3. Clustered page tables — one page table entry for several physical pages.
4. Inverted page tables — basically a coremap with a hash chain to find the translation from virtual to physical pages. See next page.

Inverted Page Tables

- An inverted page table has one entry per physical page, and contains information about which virtual page it is, and who owns it.
- To find out which physical page a virtual page is stored in, the inverted page table is organized as a hash table.
- There is one such table in the system in contrast to one normal page table per process. However, to find a page not currently in memory, we do need a normal per process page table. This table can be stored in virtual memory since it is not needed so often.
- More complex to implement shared pages since page table entry says which process owns the physical page.
- Used by IBM's UNIX System V version called AIX.

Segmentation

- Instead of linear addresses, with segmentation, an address is a pair of a **segment number** and an **offset**.
- Segmentation is normally (e.g. on x86) implemented with paging:
  - 4KB pages and up to 16K segments per process (Linux/x86 uses only six segments however).
  - The segment number is 16 bits: 13 bits index, 1 bit local/global, 2 bits protection.
  - Of the two protection bits, Linux uses one: kernel vs user mode.
  - The 13 bits index can address 8K segments.
  - There is a **Local Descriptor Table** and a **Global Descriptor Table** where each entry contains a base address and a size limit.
  - The base address is added to the offset to form a linear address which is then translated using a two-level page table.

Fork

- When a new process is created, it is (in UNIX) a copy of the parent process.
- Usually the first thing the child does is to start executing a different program.
- Physically copying the parent during fork is therefore often wasted.
- Instead the page tables of both the parent and child can point to the same physical page, and be made readonly.
- The first who wants to modify the page must first copy the page and modify the new page instead. This is called **COW** or copy-on-write.
- Possible development in Linux is to make also the page table entries copy-on-write — any volunteer for an interesting exjobb?

Page Replacement Algorithms

- When a process needs a new physical page, we might need to take one currently in use by that or another process. Being smart here is extremely important since page faults are very expensive.
- The simplest (and very stupid) method is to use reuse physical pages in a FIFO order. Simply have an index which is incremented modulo the number of pages.
- An impossible to implement but **optimal algorithm** takes the page which will not be needed in the near future. This can be used as a benchmark: if our algorithm performs within 99.9% of the optimal we cannot expect to improve our algorithm very much. Using an optimal algorithm as a benchmark is often a very useful in numerous circumstances, e.g. the SGI compiler team once discovered that their software pipeliner performed almost optimally so they were happy and could spend time on other things...

LRU

- Taking the **least-recently used** page has turned out to be a good approximation of the Optimal algorithm. This is used for many replacement algorithms: TLB, cache, virtual memory, disk cache, etc.
- However, for virtual memory it is difficult to implement:
  - We can time-stamp every page when it is accessed and then search for the least recently used page. Bad time-consuming idea for virtual memory since there are so many pages — it works for hardware caches though.
  - We can maintain a stack in software: when a page is accessed it is moved to the top of the stack. The page at the bottom is the least recently used. Horrible idea to update pointers *at every memory access instruction* — works for disk caches though. Not worthwhile to implement in hardware.

Approximation of LRU

- The **Second-chance page replacement algorithm** approximates LRU.
- Each page table entry has a **referenced bit** which indicates whether the page was referenced since it was cleared the last time by the kernel.
- Setting this bit can be done by hardware or by taking an exception to let the kernel set the bit: e.g. mark the TLB entry as invalid to cause the exception at the next access.
- Pages are scanned in a FIFO order but if their Ref bit is set, it is cleared and they get a second chance.
- If the Ref bit was not set that page is taken to the pool of free pages, but the contents is remembered just in case it is needed soon.

The Clock Algorithm

- The Second-chance algorithm is often called the clock algorithm.
- A problem with large physical memories (actually a large number of pages) is that many referenced bits tend to be set and then the clock algorithm degenerates to FIFO.
- If we instead use two clock arms:
  - one to clear reference bits, and
  - another to check them, then

  by adjusting the distance (measured in pages) between the clock arms, the algorithm can overcome this problem.

Page Table Entries

- If the page table entries also have a **Modified bit** then we get four classes of pages:

Referenced	Modified	Comment
false	false	Ideal candidate for replacement.
false	true	Second best candidate.
true	false	Will probably be used soon again.
true	true	Bad candidate for replacement.

- *Explain how the false/true case can happen!* (not very difficult...)

Page Faults

- When a process gets a page fault, it is not a good idea to start looking for a page to replace.
- Instead, the kernel keeps a pool of free pages so that one can get one quickly.
- One can also keep a pool of modified pages and write them out while the paging disk is idle.
- UNIX remembers the contents of the free pages and checks the pool first when a page fault happens.

ISA and VM

- The instruction set architecture (ISA) determines the absolute minimum number of pages that a process must be allocated — RISCs need two (one for the instruction and one for possible data accessed)
- Machines e.g. IBM 370 which can do memory copy in hardware and may need more (the IBM 370 MVC instruction may need eight pages).
- Usually it is best to let all processes compete for all the pages — this is called **global allocation**.
- If a process has too few pages in memory it will suffer from too frequent page faults. With too many such processes in memory, they will steal from each other (and themselves) and the system basically halts. This is called thrashing and is solved by swapping out entire processes.

Memory Mapped Files

- To avoid the system call overhead when reading/writing/seeking (i.e. moving to another position in the file) one can **memory map** a file.
- The file (or parts of it) will then be accessible using memory accesses so read/write a word take only one machine instruction. Seek is simply a user-mode pointer increment (also one instruction).
- The data structures and algorithms for virtual memory can be used for buffering files this way.

Page Size

A large page size leads to:

- Smaller page table.
- Internal fragmentation since not all of the page might be used.
- More efficient I/O with one large rather than many small requests are used.
- Higher TLB hit-rate since the TLB reach (pagesize $\times$ number of TLB entries) is increased. The TLB reach can be increased by using a variable page size.

Linux Memory Zones

- Three types of physical memory zones:
  - Pages that can be used for DMA: **ZONE_DMA**
  - Normal pages: **ZONE_NORMAL**
  - Temporarily mapped pages at high addresses: **ZONE_HIGHMEM**
- The details of each zone is specific to the architecture and differs between x86 and Power for instance
- On x86 DMA uses the first 16 MB
- The zones use the buddy memory allocator described earlier and allocate a number of full pages, i.e. the queue at k holds blocks of 2^k pages

Linux Memory Usage

- Three kinds of usage:
  - By the kernel: always in memory
  - Memory map: also always in memory: see below
  - The rest: including text, data, stack, page tables

Data Structures

- A 32 byte **page descriptor** contains
  - a pointer to the address space the page belongs to,
  - pred and succ pointers to make up a list (e.g. of free pages in the buddy allocator)
  - some other attributes
- **mem_map** is an array of page descriptors (less than 1% of all memory)
- The mem_map is equivalent to the **coremap** in Lab 3.
- A **node descriptor** is used for **NUMA** multiprocessors (NUMA = non-uniform memory access time, i.e. a interconnection topology that is more complex than a bus)
- The purpose of the node descriptor is to help the kernel to allocate memory close to where a thread will execute to reduce cache miss performance penalties

More on Data Structures

- Linux uses a three-level page table — but there are patches for using a fourth level e.g. for x86-64.
- Recall that the kernel itself is always in RAM
- To allocate memory for a kernel module (which can be of any size) the buddy allocator is very useful

Page and Paging Caches

- The **page cache** holds physical pages and file blocks which either
  - have recently been used, or
  - predicted to soon to be needed
- The size of the page cache is determined dynamically
- With this approach the Linux kernel can allocate pages to where they are most useful
- The page cache is the set of user pages which are on their way to the swap (but may be reclaimed before being written)

The Slab Allocator

- Recall that the buddy allocator easily causes internal fragmentation: if you need 33 pages, you must request 64 pages
- The **Slab allocator** allocates memory using the buddy allocator
- Each slab is used for a certain type of object (with the same size)
- When an object is needed from a slab, it is removed
- When all objects taken from a slab are deallocated, the slab is returned (and the memory can be used e.g. for some other object type)

The **kmalloc** and **vmalloc** Functions

- When the kernel needs some memory it can request it using **kmalloc** which itself is implemented upon the slab allocator
- The **vmalloc** allocator is used for allocating virtual addresses which don't need to be contiguous in RAM
- The latter is used for loading modules but it's preferred by the kernel to use physical pages since they will not pollute the TLB.

Kernel Memory Allocation Flags

- The memory allocation functions take options specified by the type `gfp_t` and contains or-ed bit values.
- The caller of e.g. `kmalloc` tells it what it may do, including **action modifiers**:
  - `__GFP_WAIT` – may go to sleep during the allocation
  - `__GFP_HIGH` – may use emergency pools
  - `__GFP_IO` – may start disk I/O
  - `__GFP_FS` – may start file system I/O
  - `__GFP_NOFAIL` – will retry until it succeeds
- **zone modifiers**:
  - `__GFP_DMA` – use DMA zone
  - `__GFP_DMA32` – use DMA32 zone
  - `__GFP_HIGHMEM` – use HIGHMEM or NORMAL zones

Type Flags

- The action and zone modifiers are combined into commonly used types, including:
  - `GFP_ATOMIC = __GFP_HIGH`
    – High priority and may not sleep; used e.g. by interrupt handlers.
  - `GFP_KERNEL = __GFP_WAIT | __GFP_IO | __GFP_FS`
    – Normal allocation which may block.
  - `GFP_USER = __GFP_WAIT | __GFP_IO | __GFP_FS`
    – Allocation for user processes.
  - `GFP_NOFS = __GFP_WAIT | __GFP_IO`
    – May do I/O but not file system operations — used by file systems

Address Spaces

- The address space of a process is represented by a **mm_struct** and contains of a number of memory areas
- All virtual pages in a memory area are consecutive and have the same protection
- Examples of regions are: text, data, stack, a memory mapped file
- Each area is described by a **vm_area_struct**
- The areas of a process are represented both by a linked list and a balanced tree.
- Attributes of an area include: pageable or not, growth direction (down for stack and up for data), read/write protection, and private/shared

Private vs Shared Areas

- An area can be either private to a process or shared with others
- At fork, the kernel copies the list of memory areas.
- For copy-on-write, the pages are marked as readonly and when a write occurs the kernel sees that the memory area has write permission (i.e. if it has that) and copies the page and page table and mark the entries as read/write
- **Swap** or **backing store** for a memory areas depends on which area it is (memory mapped file or a normal memory area e.g. the stack).

Backing Store

- The **text** areas use the executable file as backing store, i.e. when a text page is paged out it is simply dropped on the floor and when needed next time it is fetched from the executable file
- a **memory mapped file** not too surprisingly uses the file
- Stack and data areas use swap space when a page is paged out
- The swap space is allocated when needed
- To find the swap space an attribute in the memory area is used
- On top of the list of memory areas the **struct mm_struct** holds additional information about all memory in an address space such as:
  - How many threads are using it?
  - Where are the page tables?

PFRA: The Linux Page Frame Reclaiming Algorithm

- This is the page replacement algorithm in the Linux kernel
- There are four classes of pages for the page daemon:
  - **unreclaimable:** pages which can never be paged out
  - **swappable:** must be written to swap before being reused
  - **syncable:** must be written if they were modified
  - **discardable:** can be reclaimed directly
- The page daemon is called **kswapd**
- The init process (number 1) starts one page daemon for each memory node (recall a memory node is for NUMA architectures)

Page Daemon: One per Memory Node

- Usually about 32 pages are reclaimed each time it is woke up
- The page daemon reclaims easy pages first:
  - Discardable and unused pages are immediately moved to the zone's freelist
  - Not recently referenced pages with an assigned backing store using an approach similar to the clock algorithm
  - Shared pages which no process seems to be using
  - Ordinary not-shared user pages (i.e. modified or without backing store)
  - Other pages are skipped: if used in DMA transfer, shared and used, locked
- Shared pages require more work to page out since the page tables of all processes sharing the page must be updated

Active and Inactive Lists

- Each page is on either an **active** or **inactive** list
- When the page daemon finds a page whose referenced bit was zero that page is moved to the inactive list
- Instead of the normal clock algorithm's two states (referenced true or false), Linux uses four states combining the referenced and an active flag