CSSE2310 — 6.1

Virtual memory

Abstraction of memory

We have assumed that there is a way for processes to have their own view of memory.

We'd like:

- Protection
  - Other processes should not be able to interfere with another process.
- Sharing
  - For interprocess communication (IPC)
  - Avoiding redundancy ("many `bash` left side — handle it")
- Relocation or Translation
  - Some variables and functions have addresses fixed at compile time.
  - Different programs?
  - Multiple processes of the same program?

Abstraction of memory

- Varying overall allocation
  - Don't want to fix memory allocation at start of runtime.
  - Want to be flexible as to how much mem is needed at different times.
- Exceeding physical memory
  - Use secondary storage to store "idle" memory.

Virtual memory

Two types of addresses:

- virtual addresses — Used by the CPU when running user processes (eg pointers in your code)
- physical addresses — Locations in physical RAM

Hardware support to allow “automatic” translation between them without the program needing to be aware of it.

Page table

- The virtual address space is divided into equal sized pieces called “pages”.
- The physical address space is divided into “frames”
- Frames and pages are the same size.
- Page size being a power of two means addresses can be easily split

Page table

A page table is a map from pages to frames.

- Each process has it's own page table.
- There does not need to be any relationship between virtual and physical layout.

Virtual	Physical
...	
20	120
21	122
22	95
23	94

Virtual	Physical
...	
20	121
21	123
22	250
23	251

Note: contiguous virtual addresses[1] doesn't require contiguous physical adresses.

[1]critical for arrays

Address translation

Split a virtual address into a page number and an offset.

Hypothetical base 10 example (page size=100):

VA	Page	Offset
15609	156	9
3245	32	45
16700	167	0

- Page# = VA / pageSize
- Offset = VA mod pageSize

Address translation

page$\#|$offset $\rightarrow$ frame$\#|$offset

ie:
page$\#|$offset $\rightarrow$ PT(page$\#$)$|$offset

A note on realism

Our aim in this course is to use these calculation to help you understand the concepts.
There is no guarantee that particular combinations of params appear in physical chips.
This is also the reason we will (largely) confine ourselves to 32bit address spaces.
We will not be considering specific features such as COW or NX-protection

Implementation

Page tables are stored in memory.

- Suppose 32bit virtual address with 4kiB page size.
- A single process' page table would need: $2^{32}/2^{12} = 2^{20}$ page table entries
- For a 4 Byte PTE, this means a 4MiB table (per process).
- eg my linux desktop has 91 processes running ...

TLB

- Needing to lookup a frame in the page table means that each memory access from a program would need two memory accesses[2].
- Reduce this burden using a Translation Lookaside Buffer (TLB).
    - Caches (page$\rightarrow$frame) mappings
    - Associative memory
    - Fast
    - Hardware only goes to actual table if there is a TLB “miss”

[2]In the general case

Page faults

When there is no frame corresponding to a given page, a "page fault" occurs.

Causes:

- The page is legal for that process.
  - It has been "paged" out to disk[3]
  - The kernel needs to suspend the process until it can get that page back into RAM and find a frame to put it in.
- The page is not legal for that process[4].
  - eg: followed the null pointer.
  - eg: writing to a read only page.
  - Kernel needs[5] to inform the process (and possibly kill it).

[3] "unfortunately" called swapping sometimes.

[4] Or not legal to be accessed in that way

[5] probably

How null pointer?

- Core idea: legal vs illegal decisions are made at the level of pages not individual addresses.
- If page 0 (and possibly very low numbered pages) are always marked as invalid, then hardware makes sure the null pointer will explode.

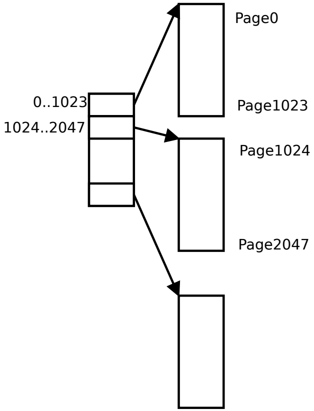
Page replacement algorithms?

Beyond the scope of this course.

Separate address spaces

- This comes as a consequence of each process having its own table.
- Shared pages happen when the kernel maps a frame in multiple page tables.
- Memory protection:
  - Pages are only valid if the kernel maps them to a frame.
  - A process can not construct a pointer to another process' frames.

Multi-level tables



Multi-level tables — gaps

Entries in the top level of the table can be "empty". That is there are no valid pages in the range for that top-level entry.
So, the table doesn't need to explictly store every entry.

User Space Memory Management

Memory layout

top of mem	
	kernel memory
	bottom of stack
	top of stack
	...
	...
	top of heap
	bottom of heap
	other data
	text “segment”
	forbidden
0	forbidden

Memory mapped content goes somewhere between heap and stack.

Lots of room in 64bit?

It's possibly quite crowded in a 32bit address space.
In a 64bit address space, there is more room. The precise location of heap and stack could vary if ASLR[6] is enabled.

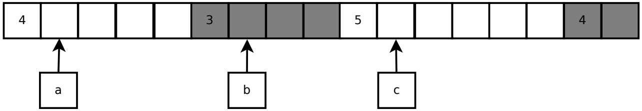
[6]Address Space Layout Randomisation

Kernel heap interaction?

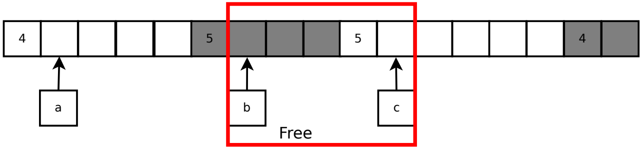
The kernel only cares about where the top of the heap is. If the heap needs more space, then a system call will allocate more valid pages to the process.
Malloc is a userspace function (which will ask the kernel for more pages if needed).

Trashing the heap

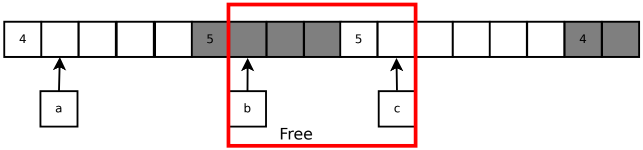
Allocators may record audit information near the allocated memory. (For example the size of the allocation).



Now consider `a[4]=5; free(b);`



Trashing the heap



- Now c[0] is in free space (as is the size of c).
- If this space is allocated to something else. `free(c)` could get “interesting”.

Memory related bugs

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

```
int val;

scanf("%d", val);
```

Reading Uninitialized Memory

Assuming that heap data is initialized to zero.

```
/* return y =Ax */
int* matvec(int** A, int* x) {
    int* y = malloc(N * sizeof(int));
    for (int i=0; i<N; ++i) {
        for (int j=0; j<N; ++j) {
            y[i] += A[i][j]*x[j];
        }
    }
    return y;
}
```

Overwriting memory

Allocating the (possibly) wrong sized object.

```
char** p;
p = malloc(N * sizeof(char));
for (int i=0; i<N; ++i) {
    p[i] = malloc(M * sizeof(char));
}
```

Overwriting memory

Off-by-one error

```
char** p;
p = malloc(N * sizeof(char));
for (int i=0; i<=N; ++i) {
    p[i] = malloc(M * sizeof(char));
}
```

Overwriting memory

Not checking string/buffer size

```
char line[8];

scanf("%s", line);          // enter 123456789....
```

Overwriting memory

Misunderstanding pointer arithmetic.

```
int* search(int* p, int val) {
    while(*p && *p != val) {
        p += sizeof(int);
    }
    return p;
}
```

Referencing nonexistent Variables

eg returning pointers to reclaimed stack memory

```
int* foo(void) {
    int val;
    return &val;
}
```

Multiple frees

```
x = malloc(N * sizeof(int));
// manipulate x
free(x);

y = malloc(M * sizeof(int));
// manipulate y
free(x);
```

Use after free

```
x = malloc(N * sizeof(int));
// manipulate x
free(x);

y = malloc(M * sizeof(int));
for (int i=0; i<M; ++i) {
    y[i] = x[i]++;
}
```

Memory leak

```
void foo(void) {
    int* x = malloc(N * sizeof(int));
    ...
    return;
}
```

Memory leak — partial free

```
struct node {
    int val;
    struct node* next;
};

void foo(void) {
    struct node* head = malloc(sizeof(struct node));
    head->val = 0;
    head->next = 0;
    ... // create and use rest of list
    free(head);

}
```

Tools

- gdb
  - Can detect symptoms
  - Doesn’t help with leaks and other problems
- valgrind
  - reports on leaks
- debugging versions of malloc

Calculations

Goal

- We want you to understand the relationship between different parameters. You should be able to answer these types of questions but please don't learn them as mechanical processes.
- Please pause the video and attempt the questions before moving on to the answers.

Example 1

Consider a multi-level page table structure:

- 32bit virtual address space
- 4-byte page table entry
- 4KiB pages

Given a multi-level page table struct as described earlier, how much memory is needed for the page table using 512MiB of *contiguous* memory?

Example 1

- Note: This is a bad question!
- Why?
    - Because it doesn't say where the memory starts.
    - Exam questions would have that information.

Let's assume it starts at zero.

Example 1

- $512\mathrm{MiB} = 2^{29} Bytes$
- How many pages is that? $2^{29}/2^{12} = 2^{17}$ pages. (Memory / page size)
- So we need 2^{17} page table entries.
- How many entries fit per page of the table? $2^{12}/2^{2} = 2^{10} = 1024$ entries per page of table.
- How many pages of table do we need? $2^{17}/2^{10} = 2^{7}$
- We need an extra page for the top level of the table.
- So to store the table we need $2^{7} + 1$ pages $= (128 + 1) * 4KiB$.

Example 2

- Given a multi-level page table struct as described earlier, how much memory is needed for the page table using 1MiB of memory?

Example 2

Assuming start address of 0 again.

- $1\text{MiB} = 2^{20}\text{Bytes}$
- How many pages is that? $2^{20}/2^{12} = 2^{8}$ pages. (Memory / page size)
- So we need 2^{8} page table entries.
- How many entries fit per page of the table?
  $2^{12}/2^{2} = 2^{10} = 1024$ entries per page of table.
- How many pages of table do we need?
  $2^{8} < 2^{10}$ So 1.
- We need an extra page for the top level of the table.
- So to store the table we need $1 + 1$ pages $= (1 + 1) * 4KiB$.

Example 3

Consider a system with 36-bit physical addresses and 32-bit virtual addresses. Pages are 8KiB (each) and page table entries are 4 bytes each.

- If the system were to use single level page tables, how much memory is used by the page table for the process?
- If the system uses two-level page tables how much memory is used by the page table for a process using 1GiB (contiguous)?

Example 3.1

- In this question, we don't need to know how big the physical address space is so ignore that value.
- To cover 2^{32} Bytes, with 2^{13} Byte pages requires 2^{19} entries.
- Each page of table can hold $2^{13}/2^{2} = 2^{11}$ entries so we need $2^{19}/2^{11} = 2^{8}$ pages $\rightarrow 2^{8} * 2^{3}\mathrm{KiB}$

Example 3.2

Again assuming starting at location 0.

- $1\mathrm{GiB} = 2^{30}$Bytes
- How many pages is that? $2^{30}/2^13 = 2^{17}$ pages.
- Each page of table can hold 2^{11} entries (last slide).
- So we need $2^{17}/2^{11} = 2^6$ pages $+$ 1 for top level
- So $65 * 8\mathrm{KiB}$

Example 4

Suppose the system has 8KiB pages and a process has the page table below:

Page#	Frame#
0	-
1	-
2	25
3	19
...	
21	18

What physical addresses do the following virtual addresses map to?

- 16735 pg=2, off=351, fr=25, addr=25*8Ki+351=205151
- 16383 pg=1, off=8191 ... SEGV
- 32768 pg=4, ... ???
- 172034 pg=21, off=2, fr=18, addr=18*8KiB+2=147458