Note that forked processes copy their parents address spaces bit-for-bit.

When using virtual address spaces, it is physically impossible for programs to access other programs memory – they will simply access null pointers in page tables.

- The advanteages of virtual address space is:
- protection: it's impossible to access other processes memory.
- You can hve less physical memory and you never need to know that.

This solves both protection and the address binding problem at once.

Load time allocation:

If you allocate memory at load time then you have to allocate the maximum amount of memory you could possibly need during process execution. This is because you can't allocate any more memory during execution. This can be very large if the amount of memory needed is dependant on inputs. For example:

```
int main()
{
        int var;
        scanf("\%d", &var);
        if (var == 1){
                int i = 2;
                int j = 3;
                int k = 4;
                printf("\%d", i + j + k);
        }
        return 0;
}
```

Note that in this situation, load-time memory allocation would allocate memory for i, j, k even if they were never used while run-time memory allocation would not need to do so unless they were required.

Addressing:

Note that you address bytes rather than bits!!

Note that when accessing page tables, you also need to access the metadata. IE you have Read-Write-Execute bits and kernel mode etc.

The more levels of a page table you have, the more time it takes to walk the whole page table. You have to walk every time there is a TLB miss.

You have to have uniformly lengthed page tables because hardware is designed to walk a specific length. So for example you could not have a page system where smaller processes had page tables of depth one and larger processes had page tables of length 3. Although the amount of disk reads would be reduced, the hardware would not support it and so the execution would be slower.

Storage:

Processes have three main segments (which we will concern ourselves with):

- The Stack Segment

- The Data Segment
- The Code Segment

The Code and Data segments are static and at the bottom of memory. The stack is at the top and "grows" downwards. This means that the empty memory is in the middle.

The stack segment grows downwards. A segment fault **on the stack** (and not in other places) can sometimes allocate memory rather than kill the process.

The CPU works with virtual addresses rather than physical addresses.

A variable can have its address changed in copy-on-write. Although it's rare.

The first thing you do after a fork is to clear the address space of the parent and load in your own parent.

Temporal locality of reference:

If something has been used recently then it is likely to be used in the near future. LRU relies heavily on the assumption of Temporal locality of reference and so can perform poorly if it is broken.

Another case where LRU performs poorly is if you ie have 5 frames and cycle through 6 pages. In this case the next process to use is the process you've just removed.

copy-on-write: Each process thinks they've got their own copy of the data. copy-on-write is a way to make copying memory more efficient by doing it lazily.

In Copy-on-write, the pages from two different address spaces are mapping to the same physical frame so they are acessing the same underlying memory without knowing.

Before a page fault, you swap to kernel mode, swap the processes CPU registers in it's PCB. You then look at the offending addresses page table entry. If it's invalid (you don't have the access) then it is a segment fault and you should kill the process. Otherwise you schedule the page to be loaded in and update the page table entry to reflect the change and then load the process which was interrupted back into the ready queue.

Why is a page fault more complex than handling an interrupt of software trap:

- Interrupts and software traps always happen between instructions (or the CPU can finish its current instruction and *then* deal with it).
- Page faults happen midway through instructions so you will have to flush the pipeline and undo whichever instruction you are midway through.
- In some cases you can have a single instruction with several memory accesses, so you will have partial results ie you have loaded two accesses and the third causes a page fault.