Address spaces and memory management

Review of processes
- process = one or more threads in an address space
- thread: stream of execution
  unit of concurrency
- address space: memory space that threads use
  unit of data

Address space abstraction
- address space: all the data the process can use as it runs. Includes program code, stack, data segment

- hardware interface (physical reality): one memory of small size, shared between processes
- application interface (abstraction provided by OS): each process has its own memory, as large as the virtual address space

Illusions provided by address spaces
- address independence: same numeric address can be used in different address spaces (i.e. different processes), yet remain logically distinct
- virtual memory: an address space can be larger than the amount of physical memory on the machine
- protection: one address spaces can't access data in another address space (actually controlled sharing)

Uni-programming

1 process runs at a time (viz. one process occupies memory at a time)

Always load process into the same spot in memory (and reserve some space for the OS)

```
fffff (high memory)
  .
  .    operating system
  .
80000
7ffff
  .
  .    user process
  .
00000 (low memory)
```

Achieves address independence by always loading process into same physical memory location

Problems with uni-programming?

Multi-programming and address translation

Multi-programming: more than 1 process is in memory at a time

- need to support address translation
- need to support protection

Must translate addresses issued by a process so they don't conflict with addresses issued by other processes

- static address translation: translate addresses before execution (translation remains constant during execution)
- dynamic address translation: translate addresses during execution (translation may change during execution)

Is it possible to run two processes at the same time (both are in memory) and provide address independence with only static address translation?

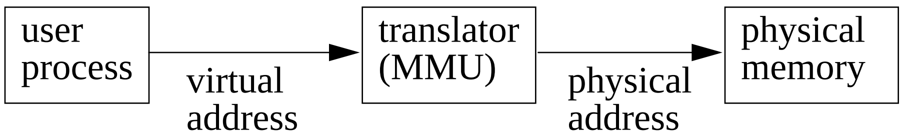
Does this achieve the other address space abstractions?

Achieving all the address space abstractions requires doing some work on every memory reference

Dynamic address translation

Translate every memory reference from virtual address to physical address

- virtual address: an address viewed by the user process (the abstraction provided by the OS)
- physical address: an address viewed by the physical memory



Translation enforces protection

- one process can't even refer to another process's address space

Translation enables virtual memory

- a virtual address only needs to be in physical memory when it's being accessed
- change translations on the fly as different virtual addresses occupy physical memory

Many ways to implement translator

Does dynamic address translation require hardware support?

Address translation

Lots of ways to implement the translator. Remember big picture:

Tradeoffs:

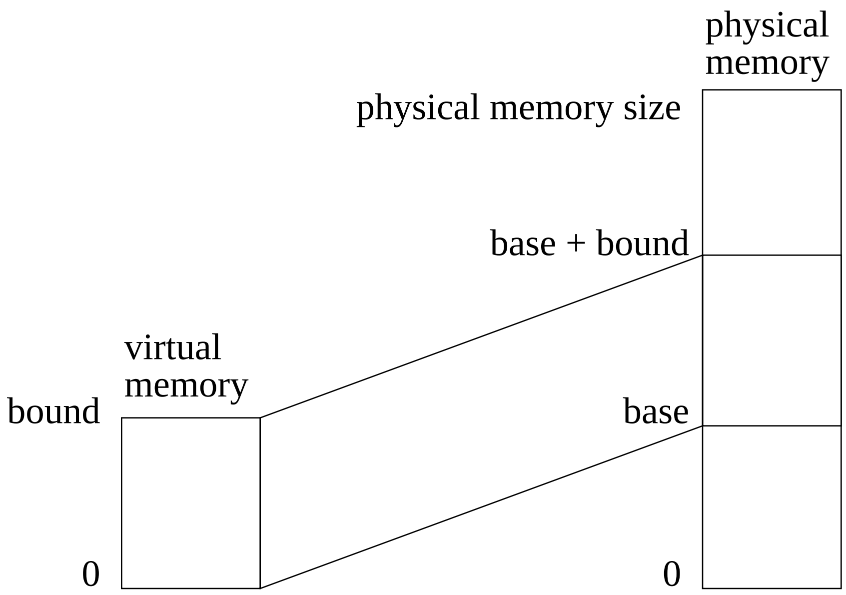
- flexibility (e.g. sharing, growth, virtual memory)
- size of translation data
- speed of translation

Base & bounds

Load each process into contiguous regions of physical memory, prevent each process from accessing data outside its region

```
if (virtual address > bound) {
   kill process (core dump)
} else {
   physical address = virtual address + base
}
```

Process has illusion of running on its own dedicated machine, with memory [0, bound)



This is similar to linker-loader, but also protect processes from each other.

As with all translation data, only kernel can change base and bounds

During context switch, must change all translation data (base and bounds registers)

What to do when address space grows?

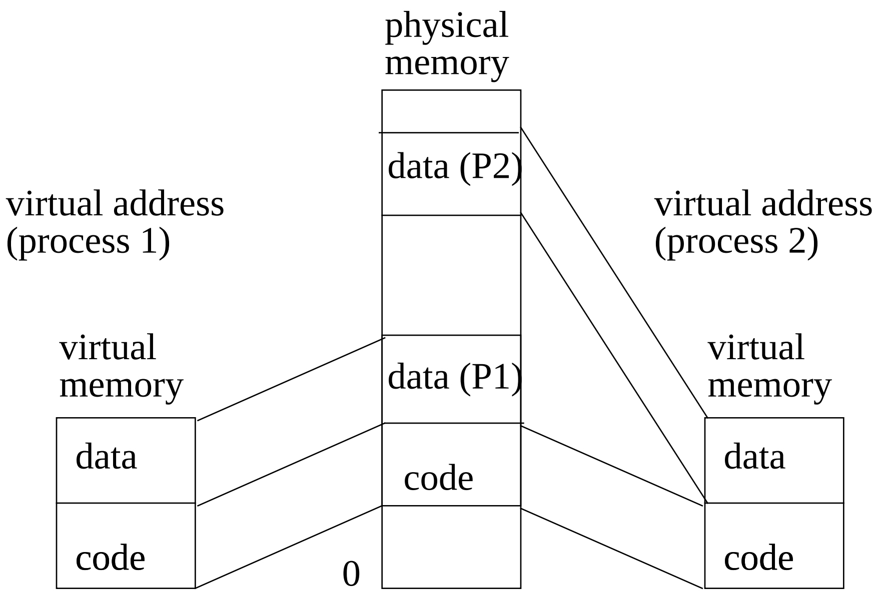
Low hardware cost (2 registers, adder, comparator), low overhead (add and compare on each memory reference)

Hard for a single address space to be larger than physical memory

But sum of all address spaces can be larger than physical memory

- swap an entire address space out to disk, swap address space for new process in

Can't share part of an address space between processes



External fragmentation

- processes come and go, leaving a mishmash of available memory regions

```
process 1 start:100 KB (phys. mem. 0-99 KB)
process 2 start:200 KB (phys. mem. 100-299 KB)
process 3 start:300 KB (phys. mem. 300-599 KB)
process 4 start:400 KB (phys. mem. 600-999 KB)

process 3 exits (frees phys. mem. 300-599 KB)
process 5 start:100 KB (phys. mem. 300-399 KB)
process 1 exits (frees phys. mem. 0-99 KB)
process 6 start:300 KB
   300 KB are free (400-599 KB; 0-99 KB), but
   not contiguous
```

- this is called "external fragmentation": wasted memory between allocated regions. Can waste lots of memory.

Allocation strategies to minimize external fragmentation

- best fit: allocate the smallest memory region that can satisfy the request (least amount of wasted space)
- first fit: allocate the memory region that you find first that can satisfy the request
- in worst case, must re-allocate existing memory regions (by copying them to another area)

Hard to grow address space

- might have to move to different region of physical memory (which is slow)
- what parts of the address space might grow as the process runs?

Segmentation

Segment: a region of contiguous memory

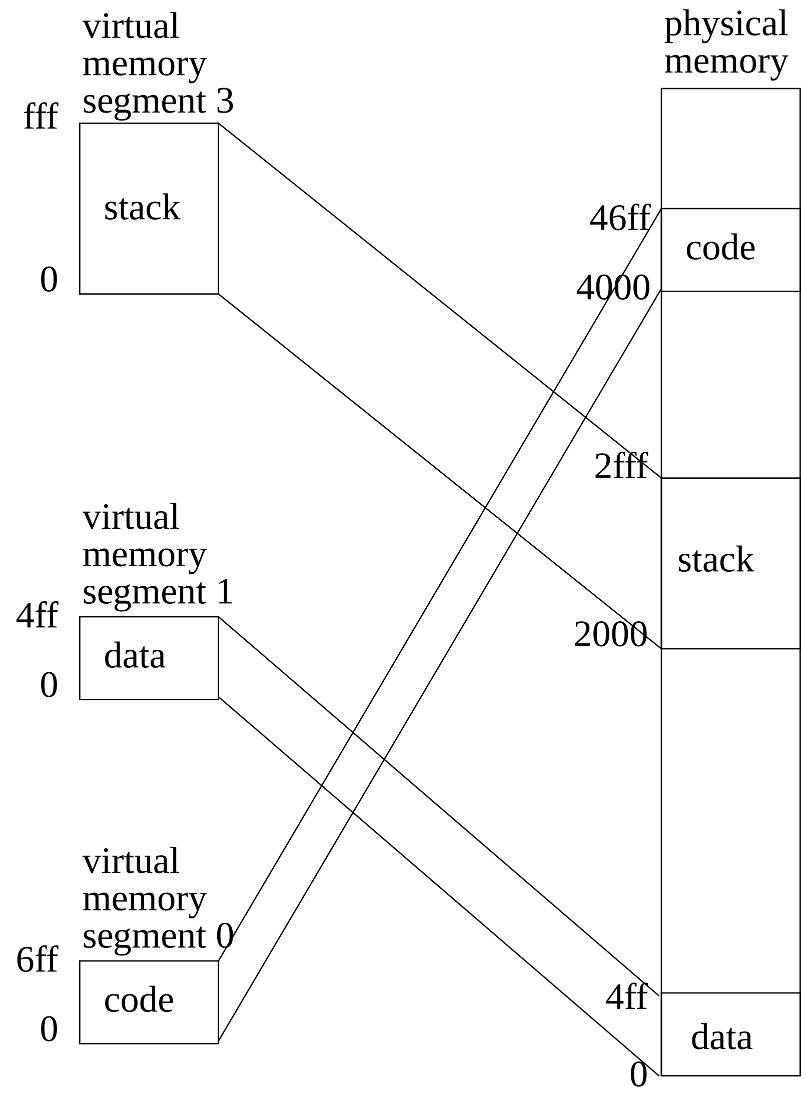
Base & bounds used a single segment. Let's generalize this to allow multiple segments, described by a table of base & bound pairs.

segment #	base	bound	description
0	4000	700	code segment
1	0	500	data segment
2	unused		
3	2000	1000	stack segment

In segmentation, a virtual address takes the form:

(virtual segment #, offset)

- could specify virtual segment # via the high bits of the address, or via a special register, or implicit to the instruction opcode



Note that not all virtual addresses are **valid**

- e.g. no valid data in segment 2; no valid data in segment 1 above 4ff
- valid means the region is part of the process's virtual address space. Invalid means this virtual address is illegal for the process to access (and will cause a core dump if accessed).
- possible to deliberately allow invalid addresses to automatically extend the address space (e.g. in Unix, accessing invalid stack address right above stack bound will trap to the kernel and automatically increase the stack size).

Protection: different segments can have different protection

- e.g. code can be read-only (allows instruction fetch, load)
- e.g. data is read/write (allows fetch, load, store)
- in contrast, base&bounds gives same protection to entire address space

What must be changed on a context switch?

Pros and cons

+ works well for sparse address spaces (with big gaps of invalid areas)

+ easy to share whole segments without sharing entire address space

- complex memory allocation

Can a single address space be larger than physical memory?

How to make memory allocation easy and allow an address space easily be larger than physical memory?

Paging

Allocate physical memory in terms of **fixed-size** chunks of memory (called pages)

- fixed unit makes it easier to allocate
- any free physical page can store any virtual page

Virtual address

- virtual page # (high bits of address, e.g. bits 31-12)
- offset (low bits of address, e.g. bits 11-0 for 4 KB page)

Translation data is the page table data

virtual page #	physical page #
0	10
1	15
2	20
3	invalid
...	invalid
1048575	invalid

Translation process

```
if (virtual page is invalid) {
   trap to OS fault handler
} else {
   physical page # = pageTable[virtual page #]
                         .physPageNum
}
```

What must be changed on a context switch?

Each virtual page can be in physical memory or paged out to disk (just like segments could be “swapped” out to disk)

How does processor know that virtual page is not in physical memory?

Like segments, pages can have different protections

- e.g. read, write, execute

Valid vs. resident

Resident means a virtual page is in memory. It is NOT an error for a program to access a non-resident page

Valid means a virtual page is not currently legal for the program to access

Who makes a virtual page resident/non-resident?

Who makes a virtual page valid/invalid?

Why would a process want one of its virtual pages to be invalid?

Page size

What happens if page size is small?

What happens if page size is really big?

Could we use a large page size but let other processes use the leftover space in the page?

Page size is typically a compromise, e.g. 4 KB or 8 KB

Fixed vs. variable size partitions

- fixed size (pages) must be compromise (e.g. 4 or 8 KB). Too small a size leads to a large translation table, while too large a size leads to internal fragmentation
- variable size (segments) can adapt to the need, but it's hard to pack these variable size partitions into physical memory (leading to external fragmentation)

What happens to paging if the virtual address space is sparse (most of the address space is invalid, with scattered valid regions)?

Paging pros and cons

+ simple memory allocation
+ can share lots of small pieces of an address space

+ easy to grow the address space. Simply add a virtual page to the page table, and find a free physical page to hold the virtual page before accessing it.

- big page tables

Comparing basic translation schemes

- base&bound: unit of translation (and swapping) is an entire address space
- segments: unit of translation (and swapping) is a segment (a few large, variable-sized segments per address space)
- page: unit of translation (and swapping/paging) is a page (lots of small, fixed-sized pages per address space)

How to modify paging to take less space?

Multi-level translation

Standard page table is a simple array (one degree of indirection). Multi-level translation changes this into a tree (multiple degrees of indirection).

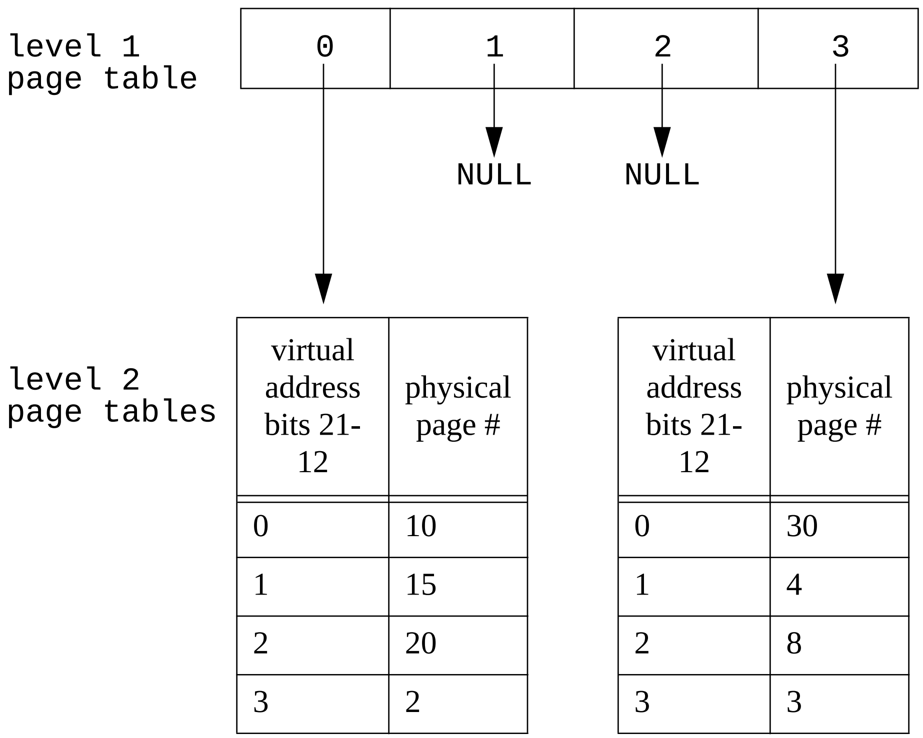
E.g. two-level page table

- index into the level 1 page table using virtual address bits 31-22
- index into the level 2 page table using virtual address bits 21-12
- page offset: bits 11-0 (4 KB page)

What information is stored in the level 1 page table?

What information is stored in the level 2 page table?

This is a two-level tree



How does this allow the translation data to take less space?

How to use share memory when using multi-level page tables?

What must be changed on a context switch?

Another alternative: use segments in place of the level-1 page table. This uses pages on level 2 (i.e. break each segment into pages)

Pros and cons

+ space-efficient for sparse address spaces
+ easy memory allocation
+ lots of ways to share memory
- two extra lookups per memory reference

Translation lookaside buffer (TLB)

Translation when using paging involves 1 or more additional memory references. How to speed up the translation process?

TLB caches translation from virtual page # to physical page # (TLB conceptually caches the entire page table entry, e.g. dirty bit, reference bit, protection)

If TLB contains the entry you're looking for, can skip all the translation steps above

On TLB miss, figure out the translation by getting the user's page table entry, store in the TLB, then restart the instruction.

Does this change what happens on a context switch?

Replacement

One design dimension in virtual memory (and any cache) is which page to replace (i.e. evict) when you need a free page

Goal is to reduce the number of page faults

Random replacement
- easy to implement, but poor results

FIFO
- replace the page that was brought into memory the longest time ago
- unfortunately, this can replace popular pages that are brought into memory a long time ago (and used frequently since then)

OPT
- replace the page that won't be used for the longest time
- this yields the minimum number of misses, but requires knowledge of the future

LRU (least recently used)
- use past references to predict the future (temporal locality)
- if a page hasn't been used for a long time, it probably won't be used again for a long time
- this yields low miss rate (similar to OPT), but is hard to implement exactly

- LRU is an approximation to OPT. Can we approximate LRU to make it easier to implement without increasing miss rate by too much? Basic idea is to replace an old page (not necessarily the oldest page).

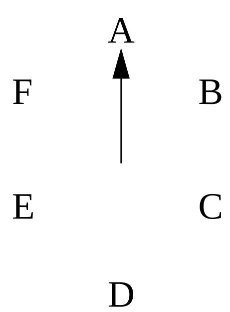
Clock

Most MMUs maintain “referenced” bit for each resident page, which is set automatically when the page is referenced. Reference bit can be cleared by OS.

Why is hardware support needed to maintain the reference bit?

How can you identify an “old” page?

Try to do this work incrementally (rather than all at once)



To find a page to evict:

- look at page being pointed to by clock hand
- reference=0 means page hasn’t been accessed in a long time (since last sweep), so this is your victim.
- reference=1 means page has been accessed since your last sweep. What to do?

Can this infinite loop? What if it finds all pages referenced since the last sweep?

New pages are put behind the clock hand, with reference=1

Pageout

What to do with page when it's evicted?

Why not write pages to disk on every store?

While evicted page is being written to disk, the page being brought into memory must wait

- may be able to reduce total work by giving preference to dirty pages (e.g. could evict clean pages before dirty pages)
- if system is idle, might spend time profitably by writing back dirty pages

Page table contents

Data stored in the hardware page table
- resident bit: true if the virtual page is in physical memory
- physical page # (if in physical memory)
- dirty bit: set by MMU when page is written
- reference bit: set by MMU when page is read or written
- protection bits (readable, writable): set by operating system to control access to page. Checked by hardware on each access.

MMU (memory management unit) of the CPU is responsible for checking if the page is resident, checking if the page protections allow this access, and setting the dirty/reference bits
- if page is resident and access is allowed, then MMU translates the virtual address into a physical address (using info from the TLB and page table) and issues the physical memory address to the memory controller
- if page is not resident, or protection bits disallow the access, the MMU generates an exception (page fault)

Operating system maintains additional information for each virtual page
- disk block # (if on disk)
- which virtual pages are valid

Do we really need hardware to maintain a “dirty” bit?

How to reduce # of faults required to do this?

Do we really need hardware to maintain a “reference” bit?

Kernel vs. user mode

Who sets up the data used by translator?

Kernel is allowed to modify any memory (including translation tables)

How can kernel refer to translation table? Translation table is not really in any process's address space. It is often in physical (i.e. untranslated) memory.

- kernel can issue untranslated addresses (i.e. bypass the translator)
- kernel can map physical memory into a portion of its address space

How does machine know that the kernel is running?

- machine must know to allow kernel to bypass translator, and to allow kernel to execute privileged instructions (e.g. halt, I/O)
- need hardware support: two processor modes (kernel and user)

How have we handled the problem of protection so far?

- implement protection by translating all addresses. But who can modify data used by translator?
- only kernel can modify translator's data, but how does processor know if kernel is running?
- mode bit distinguishes between kernel and user. But who is allowed to modify mode bit?

Switching from user process into kernel

What causes a switch from a user process into the kernel?

Sequence of events that take place when C++ program calls cin

- C++ code calls cin
- cin is a standard library function that calls read()
- read() is a standard library function that executes the assembly-language instruction “syscall”, with parameters (SYS_read, file number, size) in registers or on the stack
- **when processor executes “syscall” instruction, it traps to the kernel at a pre-specified location**
- kernel syscall handler receives the trap, and calls the kernel’s read() function

Details of what happens when trapping to kernel

- set processor mode bit to kernel
- save current registers (SP, PC, general purpose registers)
- set SP to the kernel’s stack
- change address spaces to the kernel’s address space (by changing some data used by the translator)
- jump to kernel exception handler

Does this look familiar?

How does processor know exception handler’s address?

Passing arguments to system call (and getting return values)

- can store arguments in registers or memory (according to agreed-upon convention)
- if pass arguments via memory, which address space holds the arguments?

- how does kernel access user's address space?

- kernel cannot assume arguments are valid. It must be paranoid and check them all. Otherwise process could crash kernel with bogus arguments.

Process creation

Steps in creating and starting a process

- allocate process control block
- read code from disk and store into memory
- initialize machine registers
- initialize translator data, e.g. page table and PTBR
- set processor mode bit to "user"
- jump to start of program

Need hardware support for last few steps

- otherwise processor executing in user mode can't access the kernel's jump instruction

Switching from kernel to user process (e.g. after a system call completes) is the same as last 4 steps above

Multi-process issues

How to allocate physical memory between processes?

- resource allocation is an issue whenever sharing a single resource among multiple users (e.g. CPU scheduling)
- often a tradeoff between globally optimal (best overall performance) and fairness

Global vs. local replacement policy

- global replacement: consider all pages equally when looking for a page to evict
- local replacement: only consider pages belonging to the process needing a new page when looking for a page to evict. But how to set the # of pages assigned to a process?
- generally, global has lower overall miss rate, but local is more "fair"

Thrashing

What would happen with lots of big processes, all actively using lots of virtual memory?

Usually, performance degrades rapidly as you go from having all programs fit in memory to not quite fitting in memory. This is called "thrashing".

Average access time = hit rate * hit time + miss rate * miss time

- e.g. hit time = .0001 ms, miss time = 10 ms
- 100% hit rate: average access time is .0001 ms
- 99% hit rate:
- 90% hit rate:

Solutions to thrashing

- if a single process is actively using more pages than can fit, there's no solution—that process (at least) will thrash
- if problem is caused by the combination of several processes, can alleviate thrashing by swapping all pages of a process out to disk. That process won't run at all, but other processes will run much faster. Overall performance improves.

Working set

What's meant by a process "actively using" a lot of virtual pages?

Working set: all pages used in last T seconds (or T instructions)

- larger working set ==> process needs more physical memory to run well (i.e. avoid thrashing)

Sum of all working sets should fit in memory, otherwise system will thrash

- only run a set of processes whose working sets all fit in memory (this is called a "balance set")

How to pick T? What does larger T mean?

How to measure the size of a process's working set?

Examples of process creation

Unix separates process creation into two steps

- Unix fork: create a new process (with one thread). Address space of new (child) process is a **copy** of the parent process
- Unix exec: overlay the new process's address space with the specified program and jump to its starting PC (this loads the new program)

E.g. parent process wants to fork a child to do a task. Any problem with having the new process be an exact copy of the parent?

Why does Unix fork copy the parent's entire address space, just to throw it out and start with the new address space?

- Unix provides the **semantic** of copying the parent's entire address space, but does not **physically** copy the data until needed
- separating fork and exec gives maximum flexibility for the parent process to pass information to the child
- common special case: fork a new process that runs the same code as parent.

Alternative: Windows creates new processes with a single call (CreateProcess)

- Unix's approach gives the flexibility of sharing arbitrary data with child process
- Window's approach allows the program to share the most common data via parameters

Implementing a shell

Shell provides the user interface (sh, csh, tcsh, bash, zsh, etc.).
Windows Explorer is similar.

- looks like part of the operating system, but we now know enough to write a shell as a standard user program

How to write a shell?