const FilterScreen = (props) => {

  const searchparam = {searchTerm:props.route.params.searchTerm}

  const [params, setParams] = React.useState(searchparam)

  const [likeNew, setLikeNew] = React.useState(false)

  const [vGood, setVGood] = React.useState(false)

  const [good, setGood] = React.useState(false)

  const [fair, setFair] = React.useState(false)

  const [white, setWhite] = React.useState(false)

  const [black, setBlack] = React.useState(false)

  const [brown, setBrown] = React.useState(false)

  const [rating90, setRating90] = React.useState(false)

  const [rating80, setRating80] = React.useState(false)

  const [rating70, setRating70] = React.useState(false)

Tf is that

1a)

struct sem\_t {

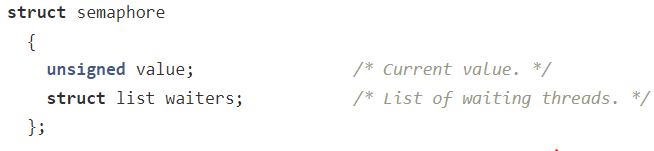
volatile unsigned value;

struct list waiters;

}

Atomic Counter that indicates the number of threads still allowed to acquire the sema. Has a queue of blocked processes.

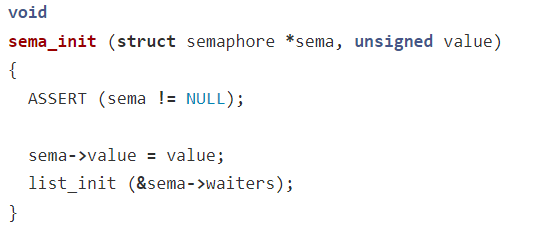
Semaphore definition in Pintos:



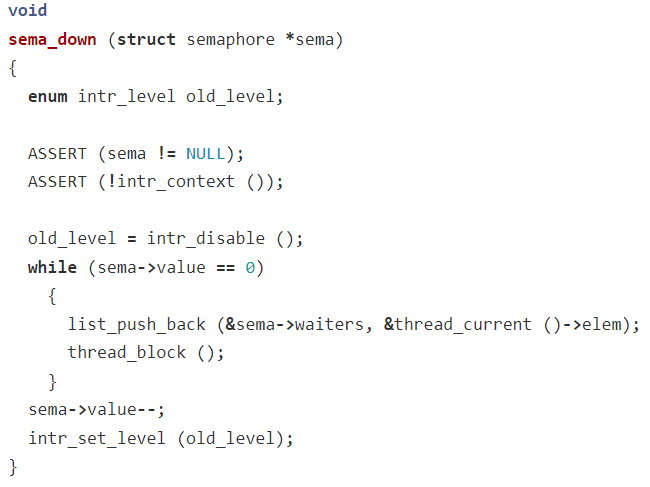
1b)

Init -> Initialise a sema at the given reference address with count value i.

The initial value indicates the number of threads that are permitted to enter the critical section.

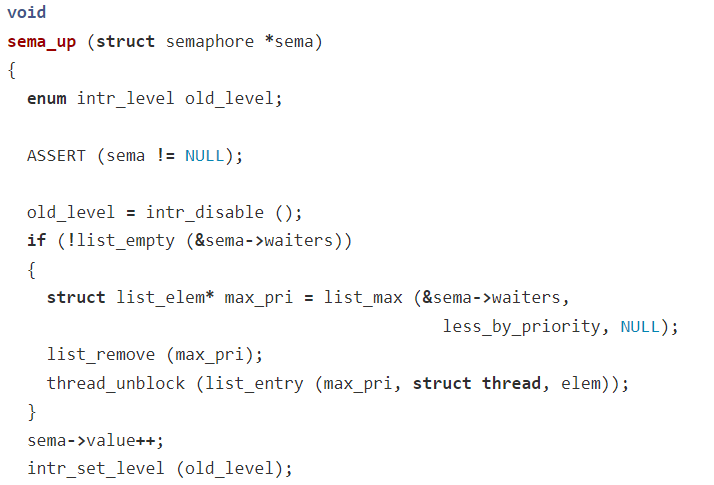


Down -> If the value == 0 then add the current thread to the waiters, then block.  
Once unblocked (if unblocked), decrement the value



.

Up -> Unblock a waiting process if the queue is non-empty, else increment the value.



These are all atomic operations.

1c) Producer-consumer 3 semaphore implementation

1 static struct sem\_t s; <- this should be initialised with value 1, but where??

2 struct semaphore \*writable = new\_semaphore(BUF\_SIZE)

3 struct semaphore \*readable = new\_semaphore(0)

4 struct semaphore \*mod = new semaphore(1)

5 void writer(int value) {

6 sema\_down(writable)

7 sema\_down(mod)

8 buf[next\_write] = value

9 next\_write = (next\_write + 1) % BUF\_SIZE,.m

10 sema\_up(mod)

11 sema\_up(readable)

12 }

13

14 int reader(void) {

15 sema\_down(readable)

16 sema\_down(mod)

17 Int r = buf[next\_read]

18 Next\_read = (next\_read + 1) % BUF\_SIZE

19 sema\_up(mod)

20 sema\_up(writable)

Return r;

21 }

There’s a solution + explanation in Tannenbaum, chapter 2.5.2

1d)

Swap line 15 and line 16. Suppose reader calls sem\_down(mod) first and obtains mod, then it needs to be able to sem\_down(readable) to continue, but it can’t sem\_down(readable) unless writer sem\_up(readable); however, writer can’t call sem\_up(readable) unless it acquires mod first. Hence, we have a deadlock by swapping 2 lines.

1e)

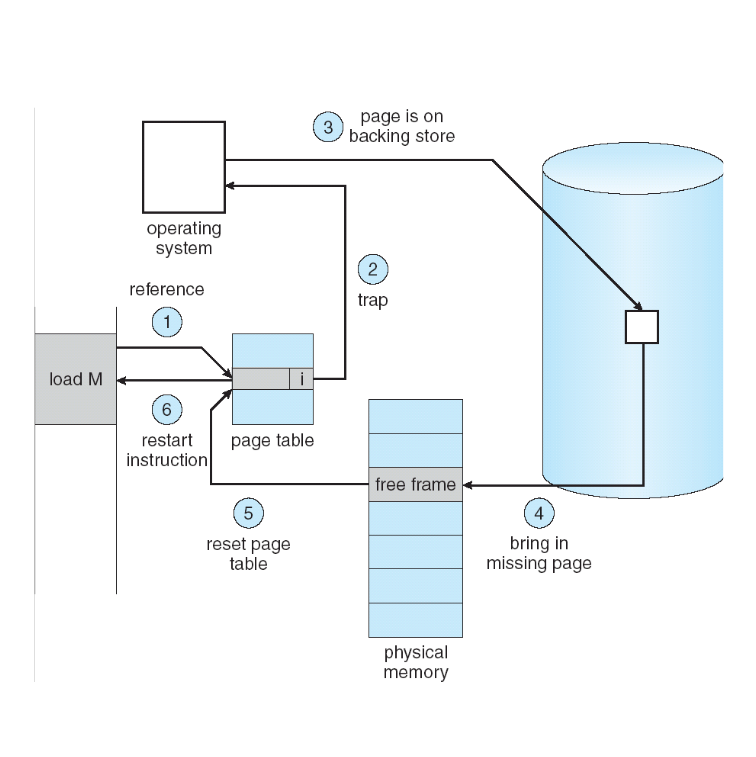
You could implement the semaphores in user-space. This is quicker because synchronisation does not require system calls.

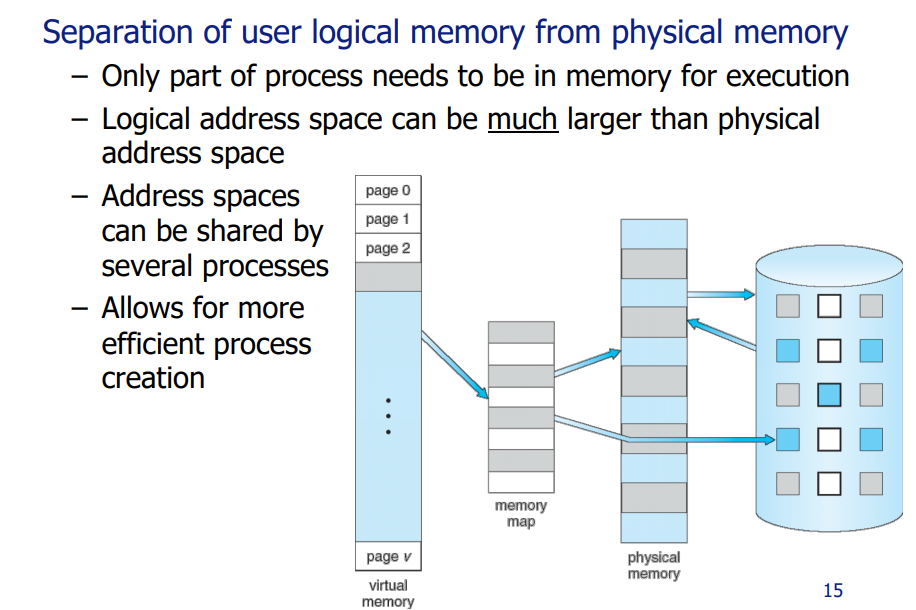
2a)

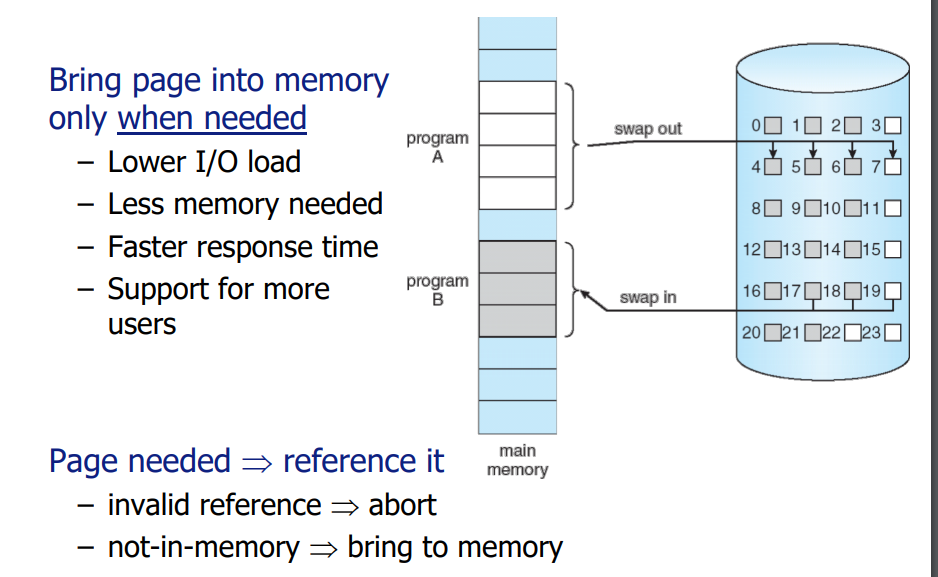
Allocation

Protection

2b)





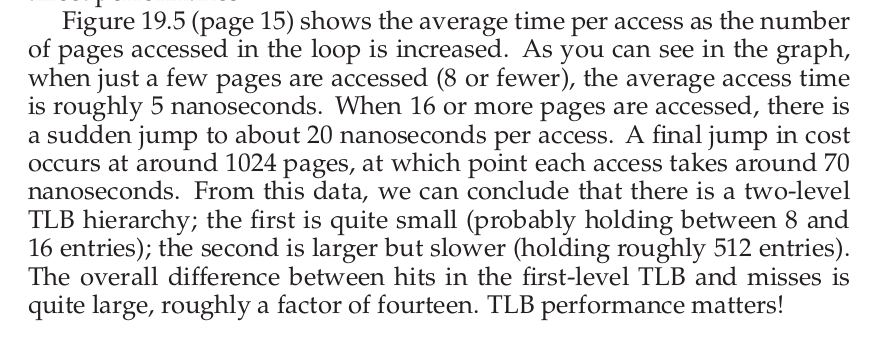


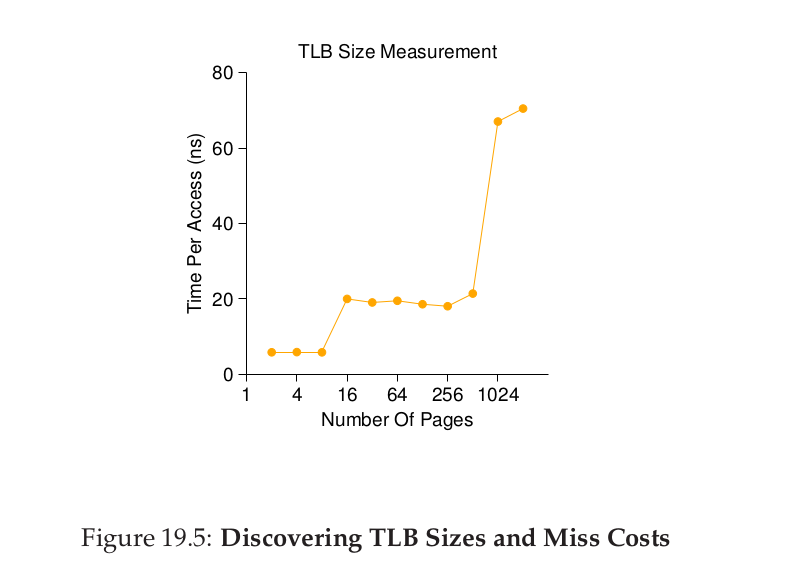
* Separation of logical and physical memory.
* Facilitated by page table which performs address translation.
* The OS manages the page table.
* The TLB is a piece of hardware that caches virtual memory lookups.
* On a context switch, the OS locates the page table for the new process, sets base and limit registers, and flushes the TLB.
* Demand paging is where you only bring in frames when they are requested.
* A valid bit is required for page table entries to keep track of pages that are and are not currently in memory.
* The OS handles eviction and page replacement policy.
* Talk about page faults, interrupts and backing store

2c i)

TLB effective access

Similar question in OS in 3 easy pieces textbook:





2cii)

Probably doesn’t use demand paging (?)

The logical address space is larger than the (user) physical address space

Eviction takes approximately 50ns?

TLB appears to have around 800 entries (1 level TLB access).

TLB eviction algorithm is shit.

2ciii)

Memory fills up quicker -> shift graph to the left, so the peak occurs earlier..

^assumes TLB is using ASID values to share TLB entries

Proportional to n, if n = 2 the peak occurs in half the time etc

^Don’t think this is correct, the graph is not against time, it is against PAGES\_ACCESSED

Alt Answer: More evictions are happening in the TLB since addresses are accessed more and in a non-deterministic fashion. So the horizontal asymptote for time would be higher. The point in which there is a sharp increase in time would be the same since the TLB/Page Table would still have the same number of entries.

Alt Alt answer: the page fault rate will increase, so when the swaps occur, EAT will be much higher and so the peak should be higher

2civ)

Hardware - use a faster disk or an SSD for faster memory access or get more memory

* Increase size of TLB (More cores = more TLBs?)
* 
* Better MMU for faster logical address translation

Software - use hashed table or inverted table for faster algorithms, use a different page replacement policy

- use a non preemptive scheduling algorithm to prevent interleaving

- use shared pages with copy on write

- use a fairer scheduling algorithm so that the same pages are accessed by each process

- Increase the number of entries in the TLB

- Buy a better PC you poor fuck