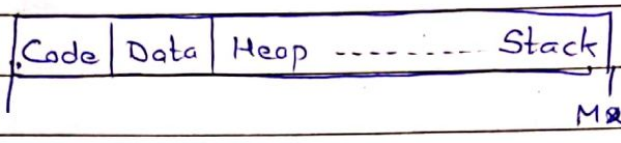
24/1

MEMORY MANAGEMENT

- Memory allocated to a process is usually not contiguous, for efficiency

- Every process believes it has memory $\in [0, \text{some max value}]$
  - 'Virtual Address Space'



- Virtual address = What programmer perceives
  eg - What &x will print
  ∴ CPU also deals with virtual addresses only

- OS must maintain a map : Virtual addresses → Real addresses
  - Map is stored on PCB.

→ Memory Management Unit. (MMU)
- Performs address translation. Present between CPU and RAM.
- Input :- Virtual address from CPU
  Output :- Real address to RAM
- OS manages, modifies MMU

CPU - Cache - MMU - RAM

→ Size of virtual address space
- Dictated by number of bits in PC

eg 32-bit architecture has 32-bit PC. ⇒ 2^{32} bytes of virtual memory (4GB)

→ Granularity of chunks that virtual memory is divided into, each of which gets a contiguous physical address space.

1 Segmentation. (→ Code, data, etc)
- Divides virtual memory into segments, each of which is mapped contiguously to physical memory

Global variables :- Data ~~(thread)~~ section
Local variables of a function - Stack
Dynamic variables - Heap

Page No.
Date

- Maintains a table, which is given to MMU

	Size	Virtual base (start)	Physical base (start)
Code segment	50	100	3000
Data segment	50	150	6000

Size is fixed

- ∴ Sizes of code segment, data segment, etc are different for different processes and may not be equal to allocated size.

Better 2 Pages

- Divides virtual memory into 'logical pages' of equal size, without caring if it is code, data, etc. and maps them to 'physical frames' of same size in physical memory.
- Granularity of allocation is 'page'
  - Some wastage of space, as each process might not use the entire page
    'Internal fragmentation' IDGAF
  - * Wastage caused by variable segment sizes in segmentation was the OS's headache
    'External fragmentation' ∴
- 'Page Table' : Logical Page No. → Physical Frame No.
  - Stored in PCB, given to MMU.
- Typical page size = 4kB

→ Expansion of a process's allocated memory
- During fork, child is allotted same memory as parent.

- Initially, there is empty memory between heap and stack.
  - Stack is automatically built up by OS.
  - System call must be made to increase heap size.
    'brk() or sbrk()'

PTO.

Kaushal

- Kernel code is not a part of C++ executable, despite being in virtual addr space of process

Page No.
Date

* 'Program break' ≜ Code + Data + Heap = in heap. Last address value
  - brk() or sbrk() expands program break (expands heap)
  - Expansion of heap is in granularity of one page.
- There actually is large separation between heap and stack.

* mmap() :- System call which returns address of any one empty page from anywhere in between (between heap and stack).

ADDRESS TRANSLATION

1/2 → Kernel Code Storage

= 1 GB out of 4 GB RAM virtual address space of every process
- Stored at the beginning of RAM.
  - Even though you know physical addresses of kernel code, MMU can only take virtual address as input.
  - ∴ Kernel code must be entered into page table instead
  - ∴ Addresses 3GB to 4GB in virtual address space of every process contain entire Kernel code, including PCBs of 'all' processes, etc.
  - These addresses in every process are accessible only when in kernel mode

* It is impossible to access memory via actual physical address
  - Have to allot pages and enter into page table.

* C library is in the physical RAM. Entries are made into all processes' page table. (in 'code' section)
  - C library is a 'shared executable'.

→ MMU

1. Translates virtual addresses (requested by CPU) to physical memory
2. Checks permissions :-

eg - Code memory of process must be Read-Only

eg - Kernel code must be inaccessible in user mode

eg - Produces segmentation fault if an unallocated virtual address is requested.

eg - Virtual address beyond Virtual Space is requested :- Error

- Raises a TRAP instruction if any permission is violated, and then jumps to kernel code before returning from trap.

→ OS

1. Maintains a 'free list' - containing all physical addresses that ~~do not~~ aren't used.
2. Allocation - via system calls : fork(), exec(), etc.
3. Constructing Page Table for PCB of every process, and updating, ~~or~~ after context switch
4. Setting Page Table for MMU
   - Give MMU new Page Table after every context switch
     ↳ (starting address of PT)
     ↳ physical, not virtual

* Copy-on-Write Fork

- Don't copy immediately. Make different copy ~~for~~ for child only when one of the child and parent wants to write back some value (change some variable)
- Reading, printing etc can be done on same memory image
- Beneficial because most children want to exec() any way.

- Both processes still have different CPU context from the beginning. But p child will still point to parent's physical memory

- Implementation :-

Data of forked process is made Read-Only.

When parent/child tries to write, MMU will raise trap.

Then memory image is copied

PAGING

eg 8-bit Virtual Address Space :- Each process has virtual addresses (0, 255)

Page Size = 16 ⇒ 16 logical pages.

eg - Accessing virtual address 35 ≡ 0010 | 0011

Page No. (2) | offset (3)

- 2^k page size :- k-bit offset

6/2 - If virtual address space has 2^v bytes.
page table has 2^{v-k} logical page entries.

- i^{th} entry in page table contains :-

1) Physical frame number (PFN) - which ~~page~~ frame is allocated to i^{th} logical page.

2) Permissions - RO, R/W ....

3) Valid - if i^{th} page is being used

4) Present 5) Dirty 6) Accessed

☆ eg Page size = 4kB

Every entry in PT takes up 4 bytes.

No. of pages = $\frac{2^{32}}{4kB}$

∴ PT size = 4 bytes × 2^{32} / 4B = 4 MB

- MMU stores the physical address of start of Page Table, and not all entries

Page No. | Date

→ Outer Page Table

- Page Table is divided into chunks, because it is too big (4MB)
- Outer page table = Mapping from Chunk # → Frame #
- So now, for obtaining physical address, get frame from OPT which will give you frame # of PT, refer that to get actual physical address
  → where chunk of PT is in physical memory
- 32-bit systems work with 2 level PT's, 64-bit requires (7) levels.
  6

32 bit Virtual address space, 4 kB page size, 4B page table entry size

Page Table size = 4MB

∴ Page Table itself needs to be stored in physical memory as 4 kB chunks.

- No. of chunks = $\frac{4MB}{4kB} = 2^{10}$
- These chunks are stored in outer page tables as 2^{10} frames
- Each OPT entry is 4B.

∴ OPT size = $2^{10} \times 4B$
= 4 kB

∴ 4 kB OPT can be contiguously stored in physical memory, so we don't need other levels.

- The MMU is given the address to outermost page table.
- Address to be fetched

10	10	12

  - First 10 bits will tell which chunk of innermost page table to look at.
  - Next 10 bits tell ~~the Base of physical address~~ table ~~look~~ the entry in innermost page that will give base physical address of page

P.T.O.

CPU cache. 'TLB' Translation Lookaside Buffer (and not the actual data)

- MMU is requested only if CPU cache misses.
- MMU cache will give the stored mapping so that MMU can reach final physical address asap.

- Every time a process context switch happens, page table is switched and entire MMU cache needs to be flushed out :(
  - Some caches could store entries for more than one process (and identify mappings based on PID)

- TLB uses LRU for replacement.

* Higher Page Size :-
  - Smaller Page Tables :)
  - Higher TLB cache hit rate :)
  - Higher internal fragmentation :((wasted space within a page)

Page No.
Date

DEMAND PAGING

3/2

→ **Swap Space**

* Demand Paging – OS allots new pages to processes if they are unused.

- If computer has an unused memory left, some pages of some processes are stored in 'swap space' of ROM (disk) to clear up space in RAM.
  - Current running process cannot be in swap space.

- Every entry in Page Table contains PFN, permissions, valid bit (seen before).
  If also has a 'present' bit – '1' if that page is in memory
  '0' swap space.
  - If present == 0, CPU cannot request for that address.
  - Present bit is meaningless if valid = 0.

→ Most OS's do not immediately allocate physical frames to data pages of a process, even though a page table entry with valid = 1, present = 0 is created.
Actual physical frame is allocated (present → 1) only when address is requested for the first time.
eg – You expand heap and put a variable there (valid → 1, present = 0)
Present → 1 only when that variable is accessed.

TLB will only contain pages for whom frames have been allocated

  - When MMU tries to translate a valid but not present address, it raises a trap called 'Page Fault'
    - OS must bring the page to memory, update Page Table. P.T.O
    - Process is blocked in the mean while. Becomes Ready when page is brought to memory. Continues execution.

Access MMU.
Check if TLB can give you frame number. If yes, fetch from memory (single access)
4 If not in TLB, traverse Page Table(s) to get physical frame (requires multiple accesses)
5 If entry in PT is not present, raise a Page Fault. Then repeat.

Both data cache and TLB have >90% hit rate.

After page fault, CPU switches to kernel mode
- While page is being fetched, OS context switches to another process (Current process is blocked)
- When fetched, OS updates PT to make present '1'.

- For addressing a Page Fault, OS must bring requested page to memory
- If memory is full, a 'Victim Page' must be 'evicted' to swap space to make room for requested page. (Total 2 memory accesses)
  - Decision of victim : 'Page Replacement Policy'
    1) Random
    2) ~~LRU~~ FIFO - Could be bad because you might evict a frequently used page.
    3) LRU - Good because of locality of reference.
    4) Approximate LRU :-
       Every Page Table Entry also has an 'Accessed' bit (1 if address in that page was accessed).
       Evict the page that is not accessed.
       - Accessed bit is periodically (~ms) reset to 0 by OS
       - OS can memorize this periodic History ~~bit~~ of Accessed bit to decide which page to evict
  - Some OS's will proactively evict a few victim pages before a Page Fault occurs (saves one memory access)

Page No.
Date

* Page Table entries also have a 'dirty' bit
(1, if page has been modified and must be written back to memory)
eg - dirty = 0 for code pages that are never modified.
- If dirty == 0, page is simply erased while evicting
Dirty bit could influence victim decision.

- Some pages are not evicted :- 'Pinned pages'
eg - Kernel code that handles Page Faults
eg - Pages that have very recently been fetched from disk.

* ps command in shell shows how much virtual and physical memory a process uses

13/2

→ **Memory Allocation and Freeing**

- 'Bitmap' - A table maintained by OS, one entry for each page.
Entry = 1 if that page is busy, = 0 if not.
- Hassle to scan through and maintain :(

Better
- 'Linked List' = 'Free list' : (done by kernel)
- Every free page contains PFN of next free page.
- Begin traversing from the 'head'
- The free list need not be in sorted order of PFN
- When you allocate free page, remove it from free list and move head to its next free page.
- Newly freed pages are added to the ~~end of free list~~ beginning of linked list, and head is set to the newly added page (not added to the end because we would have had to traverse the entire list)

VM

* Cold cache misses - Initially, cache will always miss when it is still being populated slowly

Page No.
Date

→ Variable Size Memory Allocation.

- malloc allots requested number of bytes to expand heap.
  ↓ within a page

- Variable size of chunks for allocation - 'Splitting' and 'Coales

eg Initially, entire 4kB of page is free, with 'head' pointing to start of page.

int * x = malloc (20); // Allocate 20 bytes
'x' the pointer to the start of the allocated memory.

within same block

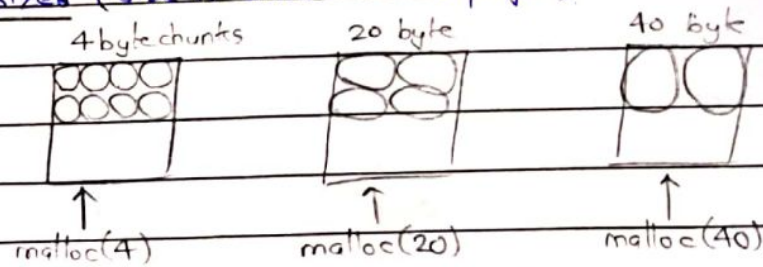
free (x); // frees up 20 bytes. Adds to free list
If not freed within that block, x will be deleted ⇒ Memory leak.

- While freeing up, free() needs to know how many bits were allotted in the first place.
  - For every allocation, malloc() stores a 'header' of h bytes to store the size of the chunk being allocated.
    same for one implementation of malloc
    - The header also contains a 'magic' bits : Random number given to that chunk before allocation
      - Can detect unintentional errors, like changing the value of 'x' or changed the size, and tried to access memory out of this chunk.

- **Coalescing** -
Periodically, consecutive small chunks in the page which are free need to be merged into a big free chunk.
Otherwise we wouldn't be able to serve a request of large no of bytes.

- If there are multiple free chunks of size ≥ requested size, which one of those should be allotted?
1) Best fit - Choose the chunk closest in size to requested size.
2) First fit - Choose the chunk that is first in free list.
3) Worst fit - Choose the largest chunk
(logic - what remains is still of respectable size)
- Actual implementation = combination of three fits.

- 'Slab Allocator' : Different implementation, - fixed sizes
Heap is made of several pages,
In this implementation, each heap page is divided into chunks of same size (size is same for a page)



- Much faster than normal malloc
- Within a page, this becomes fixed size memory allocation as before

* The kernel uses slab allocation. ~~One~~ ~~pag~~ Pages of PCB-sized chunks are used to allocate chunks for newer PCBs.

∴ Cannot allocate any other size for chunks

- Comparison

Flexibility (! Performance) : General > Buddy > Slab

variable size (user) > 2^k size (kernel) > fixed size (kernel)

15/2

TUTORIAL

Page No.
Date

Q1 Virtual Address ~ 4GB = 2^{32} bytes

Page Size = 4kB = 2^{12} bytes

Physical RAM = 8GB = 2^{33} bytes

PTE = frame # + 10 bits.

Find Page Table Size.

- No. of pages = $\frac{2^{32}}{2^{12}} = 2^{20}$
- No. of frames = $\frac{2^{33}}{2^{12}} = 2^{21}$

→ No. of bits to identify frames = 21

- Size of PTE = 21 + 10 = 31 bits

∴ 2^{20} PTEs, each is 31 bits ~ 4B

- Size required for all PTEs = $2^{20} \times 4B$
  $= 2^{22}$
  ... cannot be stored in one 4kB page
- No. of chunks of page tables in innermost level
  $= \frac{2^{22}}{4kB} = 2^{10}$
  can be stored in 4kB outer PT ✓✓
- ∴ Total PT size = $2^{22} + 4kB$
  $= 4MB + 4kB$

Q2. Virtual Address ~ 64 bits
Page Size = 4 kB
PTE = 4 bytes
Find no. of levels
Find no. of pages in PT.

- No. of pages = $\frac{2^{64}}{2^{12}} = 2^{52}$ = No. of PTE's
- ∴ Total size of all PTE's = $2^{52} \times 4$ bytes
  $= 2^{54}$ bytes.

Bytes	2	10	10	10	10	10	12
	'Level 6'					'Level 1'	

- Each 4 kB page can have 2^{10} PTE's, of 4 bytes ea

1	2^2	2^{12}	2^{22}	2^{32}	2^{42} pages

Total no. of pages = $1 + 2^2 + 2^{12} + 2^{22} + 2^{32} + 2^{42}$

* Virtual Address Space = 2^v bytes
Page size = 2^k bytes
PTE size = 2^e bytes

∴ No. of pages = 2^{v-k}
No. of PTE's for each page = 2^{k-e}

Level 1 has = $v-k$ pages
Every subsequent level has $k-e$ fewer pages
∴ No. of levels = $\left\lceil \frac{v-k}{k-e} \right\rceil$

Page No.
Date

Q3 Average memory access time = t_x

Average access time for TLB hit = t_h

miss = t_m

Find hit rate.

↳ $t_x = h t_h + (1-h) t_m$

Text

Text

Thread

on the same CPU ('Concurrency')

- 'Threads'
  - All threads of a process share the same memory image (code, data, heap, ~~stack~~) → imp
  - They run independently.

* pthreads :- Posix API for handling threads.
  eg - pthread_create(function) // new thread starts executing function
  - Multiple threads can be running at different parts of a code but be using same global variables.
  - The default thread continues execution after making new threads.

- Multiple threads can be run on different cores parallelly or time-multiplexed on a single core concurrently.
- No complex IPC-like mechanism required, as global variables are shared.

- Different threads cannot share the same stacks, as they are making different function calls.
  - ∴ Different stacks for different stacks is a necessary overhead for using threads

Page No.
Date

* Even a process can run parallelly on different cores.
  - 'Multi-core system' ≡ Cores must share same RAM.
  - Every process will have only one memory image.

→ TCB
  - Like a PCB, but has fewer things within.
    eg- Need & not store Page Table, as we can refer to parent process's page table when needed.
  - Can be included in the linked list for PCBs itself
  - Has a TID Thread identifier
    - Depending on OS, either TID or (PID of parent, TID) are unique for every thread.
  - Contains CPU context for thread (PC, registers, etc)
    * Interrupt handling mechanism is same as the one for processes. Every thread will have user mode, kernel mode, kernel stack, etc.
    * File descriptor array are the same for all threads of a process

* User-level threads
Many threading libraries give the user the illusion that there are many real threads, when in fact there are only a few actual kernel-level thread.

* pthread_join() :- Waits for thread to complete execution (like wait())

→ Issues

eg int counter = 0; // global
f() { Increment counter 1000 times }.
main() { pthread_create(f);
pthread_create(f);
}

- At the end of both threads, we expect counter to be 2000. But in practice it will be ~1800

- Reason:-

Incrementing counter (Assembly) :- load reg ← counter I1
incr reg I2
store reg → counter. I3

Problem occurs when one thread is context-switched out, when it has executed I1, but not yet executed I3.
(The other thread will read stale value of counter)

- Ideally, every load should happen only when previous store is completed.
- Need 'atomicity' - The above three statements must be executed together, if at all.
- pthread provides lock mechanism.

```
pthread_lock();
counter++;
pthread_unlock();
```

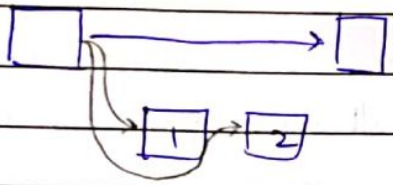
When one thread locks at one point, all further threads will block at that point until first thread has unlocked.

Page No.
Date

- 'Race condition'

When two threads try to do the same thing and one ends up overwriting the other's changes.

eg - Two threads wanting to add an element at same position in a linked list.



a.next = 1

a.next = 2

- 'Critical section' - Part of code where race condition might occur.

- 'Mutual exclusion' - When one thread is in a critical section, other thread should not be allowed entry into that section.

* Can a simple boolean variable function as a lock? (Software Lock)

No -

```
bool locked = 0;
while (locked = 1) { } ... // blocking   } 'l'
locked = 1;
--- Function ---
locked = 0;
```

problem if there is an context switch after while function. before lock=1.critical section the next thread will also execute the

Setting locked = 1 also represents a ~~race~~ critical section

More than one ~~proc~~ threads can find locked to be 0 and can both try to set it to 1 and execute the subsequent function.

Desireable :- We want the two statements in 'l' to always run 'atomically'.

If one process locks, all other processes will be blocked at 'l' until it unlocks.

* ~~Fine-grained lo~~ Granularity of lock :-
If there are 'n' shared variables, we can use 'n' different locks wherever each variable is used ('fine-grained lock') or use a big lock covering all variables.

→ Implementation
CPU provides 'Hardware Atomic Instruction' - A single instruction that loads and stores at the same time

eg - Exchange instruction in Intel
oldvalue = xchg(variablename, newvalue)

- Implementing lock :-
while(xchg(flag, 1) == 1) { }

- Even if different threads run on different cores using different caches, the 'cache coherence protocol' ensures that two threads do not access flag simultaneously.

Page No.
Date

* Compare and Swap Instruction :- CAS (variable, old value, new value)
bool x = CAS (flag, 0, 1)
x becomes 1 only if flag was zero
flag 1
- Lock :- while (!CAS (flag, 0, 1)) { }

• The above implementations are known as 'spin lock' implementations

- When one process has locked, other processes get stuck in while loop and ~~wasting~~ are wasting CPU cycles when scheduled ☹ (other processes are not blocked)

* Desirable features of lock implementation - Mutual exclusion
- Low overhead
- Fairness.

• Solution :- 'Sleeping Mutex' / 'Mutex' implementation
Use the 'yield()' function to give up the CPU.

```
while(1) {
    if (xchg(flag, 1) == 1) { yield(); }
    else { break; }
}
```

- This implementation has overhead in terms of time required to context switch
  - If the code following lock and preceding unlock is small, spin-lock implementation might be better.

* Actual pthread uses a combination of both implementation :-
Spin for a few cycles, then yield

→ Locks inside the OS.

There are many shared variables that can cause race conditions
eg - Two interrupts occur simultaneously.

- On a single-core machine,
When you are addressing one interrupt, disable interrupts until you are done. This prevents race conditions.
- Newer interrupts will queue up.

- On a multi-core machine,
We need a special locking mechanism.
(One core can disable interrupts only for itself)

* User code cannot disable interrupts.

- OS locks can only be spin-lock
- (The OS cannot 'yield' to itself)

- OS spin-locks are risky:-
- OS ~~cannot spin for a very long time~~
eg- Cannot ~~run a blocking~~ wait for disk-read.
This is because other cores can also get stuc

① OS code following lock should not be long
eg- Should not read from disk or perform blocking operation
This is because other cores will get stuck, and there is nobody to release them
(This was fine for user spin-locks, because the OS anyway context-switches after some time)

Page No.
Date

(2) After OS code locks, all interrupts must be disabled.

- One interrupt occurs. The OS locks.
  ~~At the same t~~ When second interrupt (of same type) occurs, it will keep spinning at that lock in the ISR.
  Neither of the two interrupts can be completed ^ ^ o

8/3

→ Ordering amongst threads.

- We want T1 to execute first, T2 to execute after. (some code)
  - Make a flag and make T2 spin at it till T1 completes execution
    - Wastes CPU cycles :(

Better • Condition Variable

```
pthread_cond_t cv;
```

- Maintains a queue of all threads waiting for the condition variable.
- One thread calls wait(), which blocks (instead of spinning), and the other thread executes signal() when it is done executing (signal() unblocks the first process)

T1

```
task1();
flag = 1;
pthread_cond_signal(cv);
```

T2

```
if (flag == 0) {
    pthread_cond_wait(cv)
}
task2();
```

Flag is necessary because we don't want T2 to be blocked ~~even~~ if T1 has already finished task1()

P.T.O.

- Race Condition :-

T2 checks flag and decides to block. Before it calls wait it is context-switched out and T1 is scheduled.
T1 completes all execution
Then T2 will be permanently blocked.
∴ Need to use lock

T1	T2
lock(m);	lock(m)
task1();	if (flag == 0) {
pthread_cond_signal(cv);	wait(cv);
unlock(m)	}
	unlock(m)
	task2();

Issue :- If T2 locks first and blocks on wait, it will never unlock. Both threads will be blocked.

Solution :- wait(cv, m) :- m is a mutex

① Add thread to the cv queue.
② Unlock the lock m.
③ Block till receiving signal(m).
④ Lock m again

T1	T2
lock(m);	lock(m)
task1();	if (flag == 0) {
signal(cv);	wait(cv, m);
unlock(m)	}
	unlock(m)
	task2();

* wait(cv, m) runs atomically - No race condition

☆ • Replace if by while just to be safe

→ Producer - Consumer Pattern.

- P and C are threads sharing same common buffer.
- P executes and fills buffer, then C executes and empties buffer X ∞.
- Buffer can be a single variable or an array.

P

```
{ lock(m)
  if (count==1){
     wait(cv_prod, m) }
  add item to buffer
  count++
  signal (cv_cons)
  unlock(m)
}
```

C

```
{ lock(m)
  if (count==0){
     wait(cv_cons, m); }
  consume from buffer
  count--
  signal (cv_prod)
  unlock(m)
```

13/3

→ Semaphore

sem_t s;

- Alternatively to condition variables
- Internally - As a (n integer) counter.

- Initialize — sem_init (&s, value)
- Decrement — sem_wait (&s) (OR) sem_down (&s) } Blocks if the semaphore becomes ~~zero or~~ negative
- Increment — sem_post (&s) (OR) sem_up (&s)

Page No.
Date

Initially, s = 0 (initialized in both threads before spawning)

eg

T1	T2
task1()	
up(s)	down(s)
	task2()

- Implementation of down is atomic internally.

* Semaphores can be used as a lock.
Initialize s = 1.

```
:
down(s)
//work
up(s)
```

The first thread will decrement s to zero. All subsequent threads will block on down(s). (verify) When up(s) is called, all blocked threads will be unblocked and value of s becomes 0 irrespective of ~~what~~ 'x' ~~or~~ how ~~as~~ much negative 's' was.

* Can only use up() and down() with semaphores. Cannot actually read the counter value of s in code.

→ Producer - Consumer using semaphores

CV → Semaphore

cv ~ s

wait ~ down

~~dow~~

signal ~ up

Page No.
Date

Initialize :- sem numempty = N (Size of array)
sem numfull = 0
sem m = 1

P	C
down(numempty)	down(numfull)
down(m)	down(m)
add_item()	remove_item()
up(m)	up(m)
up(numfull)	up(numempty)

- numfull indicates no. of elements produced (waiting to be consumed) in the shared buffer.
eg - 2 items in buffer
numfull = 2 (because up was called twice)
C will consume 2 items and then block on the next attempt.
- numempty indicates no. of vacant elements in buffer.

- Need to use locks for mutual exclusion of buffer accesses
- Can we interchange down(numempty) and down(m) in both P and C?
No - Deadlock ∴
∴ If P blocks on down(numempty) while holding the lock, C will block on down(m) and neither can revive the other
- Whenever you call down, ensure that the thread who will call up can actually run.

→ Synchronized Tasks.

- Neither of A → or B should start their $(N+1)^{th}$ task until both have completed their respective N^{th} tasks.

- Using condition variables,

A	B
lock(m) ← a1()	lock(m) ← b1()
a1 done = true	b1 done = true
signal (cv b)	signal (cv a)
if (! b1done)	if (! a1 done)
wait (cv a, m)	wait (cv b, m)
unlock(m)	unlock(m)
a2	b2

- Use a lock whenever there are shared variables.

- Using semaphores,

Initialize 2 semaphores :- a1 done = 0
b1 done = 0.

A	B
a1.()	b1()
up (a1 done)	up (b1 done)
down (b1 done)	down (a1 done)
a2 ()	b2.

- If up and down are interchanged in both A and B, both A and B will block at down and won't be able to revive each other.

→ Deadlocks

① Atomicity constraints :-
Hold a lock before - writing to a shared variable
- reading a shared variable that there's a chance will be written to simultaneously.

② Ordering constraints - When using condition variables or semaphores

③ Ordering of multiple locks -

T1	T2
lock(a)	lock(b)
lock(b)	lock(a)

Solution 1 :- Try to use locks in same order

lock(a)	lock(a)
lock(b)	lock(b)

Solution 2 :- Use a bigger lock over the whole piece of code.

lock(BigfatLock)	lock(BFL)
lock(a)	lock(b)
lock(b)	lock(a)
unlock(BFL)	unlock(BFL)

- We use two types of locks - one for readers, one for writers

```
int readcount;   bool writerpresent;   int writercount

Readlock()
lock(m)
if(writerpresent || writercount>0){wait(rcv, mutex)}
readcount++
unlock(m)
// read now (Critical section)

Writelock()
lock(m)
writercount++
if(readcount>0 || writerpresent){
    wait(wcv, m)}
writerpresent = true
unlock(m)
// Critical section here

Read Unlock()
lock(m)
readcount--
if(readcount == 0){
    signal(wcv)}
unlock(m)

Writeunlock()
lock(m)
writerpresent = false
writercount--
if(writercount>0)
    signal(wcv)
else
    signal(rcv)
    unlock.
    and other writers
```

- Writer can enter only when all readers have left critical section.

- Issue :- Writers won't be allowed to enter if any reader is present. Readers can enter, even if they arrived in the queue after the writer did

This implementation is called RWL with reader's preference

Page No.
Date

→ Readers - Writers Lock without readers' preference.

To block

~~Read lock()~~ ~~Read unlock~~

→ Reader's - Writers Lock using semaphores

Verify

```
Readlock()
down(m)
readcount++
if(readcount == 1){
    down(wsem)}
up(m)
    Writelock()
down(wsem)
```

```
Readunlock()
down(m)
readcount--
if(readcount == 0){
    up(wsem)}
up(m)
    Writeunlock()
up(wsem)
```

20/3

→ Barber-shop Problem

- Waiting-chairs. Occupy one if any is empty. If all are occupied, wait.
- If waiting room was empty, barber sleeps and must be woken up for a haircut.

P.T.O.

① Write count, bool variables
② Write all wait statements
③ Write all corresponding signal statements

Page No.
Date

- Simplified Barber-shop problem
  (Waiting room size = ∞)

Customer

```
lock(m)

count++
signal(barbercv)
if(barberbusy)
    wait(custcv)
getHaircut()
count--
```

Barber

```
if(count == 0)
    wait(barbercv)
barberbusy = true
GiveHaircut()
barberbusy = false
signal(custcv)
```

up(m)→

down(m)

- Unsimplified :- Waiting room size = N

Customer

```
{
    lock(m)
    if(count == N)
        unlock(m)
        exit();
    count++
    ; Same as above
    !
```

take care of initial values of semaphore

Page No.
Date

- Using Semaphores

cust = barb = 0 m =

Customer

```
down(m)
if (count == N)
    up(mutex)
    exit()
count++
up(cust)
down(barb)
// Get haircut
down(m)
count--
up(m)
```

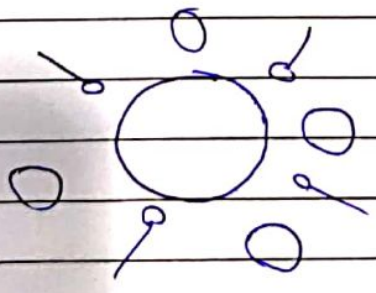
Barber

```
{
up(barb)
down(cust)
// Cut hair
```

useful, if there are no cust (to make barber wait)

No customer should hold the lock while getting a haircut. Other customers won't be able to enter waiting time.

→ Dining Philosopher's Problem.



N philosophers in a circle.
1 spoon between any two philosophers.
A philosopher can eat only after they pick up both left and right spoon.

will wait if all of them pick up [illegible] and wait on their right.

- Waiter-based solution :-
Allow philosophers to pick up both or none of the spoons at a time
- Uses a BEL

```
lock(m)
if ( right is free && left is free )
        // Pickup spoons
unlock(m)
```

- If you couldn't pick up spoons, add yourself to a queue
- After you finish eating, signal condition variables of your waiting neighbours.

→ Cigarette smoker's problem.

- 3 smokers.
- Cigarette requires 3 ingredients – Tobacco, paper, matches
  (T) (P) (M)
- Each of the smokers has one of the three ingredients.
  S1 has T
  2 P
  3 M
- Ingredients keep arriving with time.
  If M & P arrive, S1 should make a cigar and exit.
  T & P 3
  T & M 2

- If P arrives first, and S1 took it,
  Then, if M arrives next, ~~S2~~ ~~S1 and S2 should~~
  S1 and not S2 should take it
  (otherwise none can smoke).
  ∴ The following will not work

S1	S2	S3
down(p)	down(m)	down(t)
down(m)	down(t)	down(p)

- Make a combined semaphore, one for a 'pair' of ingredients.

S1	S2	S3
down(pm)	down(mt)	down(tp)