Q1 A

i) Problems / Drawbacks / limitations.

→ takes in + uses a lot of CPU time. Wastes a lot of processing time ~~to~~
→ If there is timeslicing (multi-cpu), interrupts are disabled in between the disable + enable interrupts. This can cause some other processes that rely on interrupts to not work.
→ Sleep(0) does nothing. 0 ms?

ii) ! work in multi-cpu.

→ Timeslicing + swapping in and out processes can happen during the if statement, before interrupts ~~are~~ can happen. ~~Then~~
→ May cause hanging of process

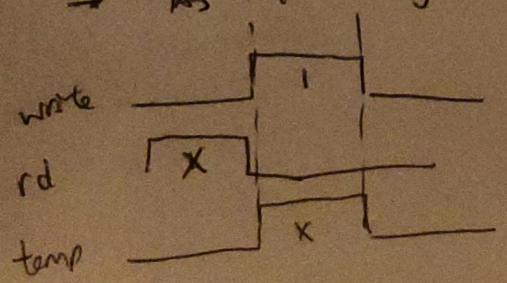
iii) TAS - atomic (Testing + Setting)

→ The TAS instruction, essentially ~~locks~~ prevents other processes from handling the memory address while it's being accessed.
→

In code: (using atomic) which uses TAS => atomic <bool> my_bool = false. <Atomic_flg?

→ Replace flag with my_bool => if (my_bool == FREE) {
my_bool = Busy;
enable_interrupts;
...
...
else {
...
}
...
~~flag~~ my_bool = FREE;

iv) Not same as ~~to~~ setting flg w/ test instructions.? TAS
→ TAS is twice as long to incorporate both reading + writing.



→ Reads value of flag
→ Sets to 1 regardless
→ Returns prev value read w/o releasing control of memory address + data buses

* Q1B

i) variables

```
bool full = false;

~~int* buffer;~~
mutex m;
semaphore s;

~~int* buffer;~~
int* buffer;

~~queue my-q;~~

string my-name;
```

ii)

```
cpipeline (string &name, int size) {
    my-name = name;
    buffer = new int[size];
}

void write (void *data, int size) {
    for(int i=0; i < size; ++i) {
        s->wait();
        m->wait();
        Data[buffer * ++] = (int) data;
    } s->signal(), m->signal();
}

void read (void *data, int size) {
    for (int i=0; i < size; ++i) {
        s->wait(); m->wait();
        (int) data = Data[buffer * ++];
        s->signal(), m->signal();
    }
}
```

iii)

```
cpipeline mypipe("exam", sizeof(int));
int main () {
    thread t1(main2); char x='x';
    while(1) {
        mypipe.write(&x, sizeof(char));
        x=getchar(); if (x==='_') break;
    }
    t1.join();
    return 0;
}

void main2() { char x='x';
    int* data;
    while (1) {
        printf("%d", mypipe.read(&data, sizeof(int)));
        if (x==='_') break;
    }
}
```

Q2 A — [illegible] interrupts [illegible] (Real time clock) [illegible]

Thread → Code (thread) written a program that runs independently. It can access the same global variables in the same process

Active OS → A class / obj that when instantiated, runs independently. Can also access some global variables in same process. It also has its own member variables, functions, and a "main" that runs automatically after the constructor.

Synchronization → A concept to 'synchronize', in time, & Synchronization is used to make sure various parallel running tasks are safe, are
Eg. Mutex, semaphore.

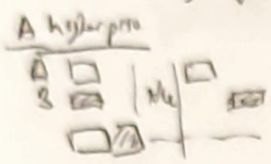
Coarse vs Fine → Granularity is essentially the degree of parallelism that exists in a system. Crude, or coarse is usually between splitting processes. Fine is more towards thread splitting and parallelly executing code. However, fine may have communication + synchronization problems.

Pre-emptive Scheduling → Pre-emptively scheduling is to essentially schedule things before they happen, based on things you know such as deadline time, trigger-period, etc. hence "pre-emptive"

Q2 B

Implemented using interrupts ~~every~~ via RTC (Real time clock) for timeslicing.

1) RMS → A type of scheduling to assign tasks based on them shortest deadline. (Theorem rule)



We're given deadline, period, and compute time. We schedule priority based on shortest deadline which means highest priority. (AKA inverse of trigger period)

2)

	period	complete
①	every 2 days	1 day
②	4	1
	7	1.5
	7	0.5

$$\frac{1}{2} + \frac{1}{4} + \frac{1.5}{7} + \frac{0.5}{7} = 0.5 + 0.25 + \frac{3}{14} + \frac{1}{14}$$
$$= 0.75 + \frac{4}{14}$$
$$= \frac{3}{4} + \frac{4}{14}$$
$$= \frac{6}{8} + 4/4 = \frac{21}{28} + \frac{x}{28}$$
$$= \frac{29}{28}$$

$$\frac{21}{28} > 100\%$$

$\boxed{\text{NO}}$

2)

4	0.75
5	1.5
7	1.25
0.5	X

$$\frac{0.75}{4} + \frac{1.5}{5} + \frac{1.25}{7} + \frac{x}{0.5}$$
$$= \frac{1.5}{8} + \frac{3}{10} + \frac{5}{28} + \frac{2x}{1}$$
$$= \frac{3}{16} + \frac{3}{10} + \frac{5}{28} + 2x < 4(2^{\frac{1}{4}} - 1)$$

75.6

66 + 27

$x < 0.0484$

3)

4	1
5	1.75
7	1.25
8	1

$$\frac{1}{4} + \frac{1.75}{5} + \frac{1.25}{7} + \frac{1}{8}$$
$$= \frac{1}{4} + \frac{7}{20} + \frac{5}{28} + \frac{1}{8}$$
$$= \frac{3}{8} + \frac{5}{28} + \frac{7}{20}$$

$= 90357$ $\quad n(2^{\frac{1}{n}} - 1)$

needs drawing.

interweight time, $\boxed{\text{YES}}$ $\boxed{\text{NO}}$

$\boxed{1}\ \boxed{1.75}\ \boxed{1.25}\ \boxed{1}$ > 4

Harmonic 9

4)

2	0.5
4	1
8	1.25
16	3

$$\frac{0.5}{2} + \frac{1}{4} + \frac{1.25}{8} + \frac{3}{16}$$
$$= \frac{1}{4} + \frac{1}{4} + \frac{37}{32} + \frac{6}{32} = \frac{2}{4} + \frac{13}{32} = \frac{16}{32} + \frac{13}{32}$$
$$= 29/32 < 100\%$$

$\boxed{\text{YES}}$ We harmonic

Q3A

Avoidance → "avoiding" the problem
→ trying not to have deadlock. (Resource request orders, giving up resources, treating multiple resources as one)
→ has drawbacks.

prevention → Preventing from happening by using Banker's alg. in the first place.
→ mitigations.

Avoidance:

1) Resource request ordering
   ↳ Acquire multiple resources in same strict order and be aware of other threads
   ↳ need designers to agree

2) Giving up resources
   ↳ After a timeout, just give it up.
   ↳ Starvation might still happen (if it give up a resource it already has it might not get any back if some other thread acquires it.

3) Treat multiple resources as one
   ↳ Just one mutex for all resources. Cannot acquire individual resource, needs all even if doesn't need all or wants just one.
   ↳ Needs thread co-operation

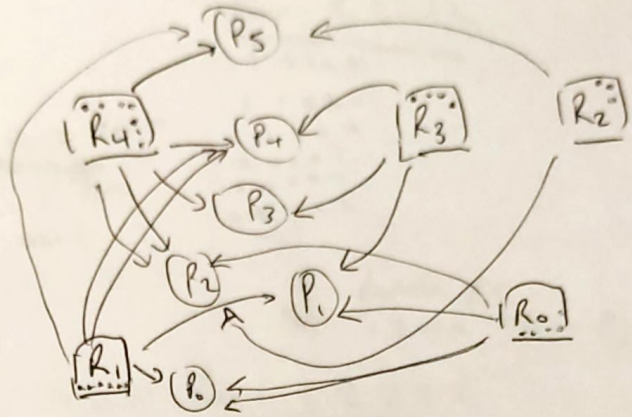
Q3B (i)

Banker's Algo

$R_4 = 7$, $R_3 = 4$, $R_2 = 3$, $R_1 = 6$, $R_0 = 5$

MAX	7	4	3	6	5
P_5	1	0	1	1	0
P_4	1	1	0	2	0
P_3	1	1	0	0	0
P_2	2	0	1	0	1
P_1	0	1	0	1	1
P_0	0	0	0	1	2
	5	3	2	4	4

start



⇓ end +

– P_5	1	2	0	0	0
– P_4	0	3	0	2	2
P_3	0	0	3	1	0
– P_2	0	2	0	3	3
– P_1	2	0	0	1	0
– P_0	1	0	2	3	2

Free: 2, 1, 1, 2, 1 → 0, 1, 1, 1, 1

① Donate all to P_1 → P_1 done

+ P_1 done

② Free: 2, 2, 1, 3, 2 → 1, 0, 1, 3, 2

Donate to P_5 → P_5 done

+ P_5 done

③ Free: 3, 4, 1, 3, 2 → 3, 1, 1, 1, 0

Donate to P_4 → P_4 done

+ P_4 done

④ Free: 3, 7, 1, 5, 4 → 3, 5, 1, 2, 1

Donate to P_2 → P_2 done

+ P_2 done

⑤ Free: 3,

See next pg!!

Q3B (1.7)

Free : 2, 1, 1, 2, 1

1	0	1	1	0
1	1	0	2	0
1	1	0	0	0
2	0	1	0	1
0	1	0	1	1
0	0	0	1	2

donate 20010 → P_1 done, free P_1

Free : 2, 2, 1, 3, 2

1	0	1	1	0
1	1	0	2	0
1	1	0	0	0
2	0	1	0	1
0	0	0	0	0
0	0	0	1	2

↓ donate P5 12000 + free P5

Free : 3 2 2 4 2

0	0	0	0	0
1	1	0	2	0
1	1	0	0	0
2	0	1	0	1
0	0	0	0	0
0	0	0	1	2

free + donate P_0 10232 →

Free : 3 2 2 5 4

0	0	0	0	0
1	1	0	2	0
1	1	0	0	0
2	0	1	0	1
0	0	0	0	0
0	0	0	0	0

↓ free + donate P2 02033

Free : 5 2 3 5 5

0	0	0	0	0
1	1	0	2	0
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

donate + free P3 00310 →

Free : 6 3 3 5 5

0	0	0	0	0
1	1	0	2	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

↓ free + donate P4 03022

Free : 7 4 3 7 5

All zero

end

Q5A

i) one way msg = uni ⟵ the arrow one calls the other

bi = two way ~~msg~~ message. ⟷ two arrows. Can call each other

ii) needs pointers (bi directional) to & from both class. Both classes need pointer to other class (member variable)

Eg. class 1 ⟷ class 2.

means ~~one~~ class1 ~~function~~ needs a * assign(class2* var) member function

OR class2 needs a assign(class1* var) member function.

ie. class 1 obj. assign (class 2ptr);

~~Inside void~~ assign (class2* ptr){ // class2memberptr is a member var in class 1.

class 2memberptr = ptr;

ptr → class1memberptr = this; // class1 memberptr is a member var in class 2

}

iii) In the obj

1 → literally just member var (singly).

n → array of member var, such as membervar[n]; and possibly a for loop to initialize ~~variable~~ inside values.

* → dynamic, using membervar* and assigning membervar = new ...;

also need to delete membervar in destructor.

iv) Aggregation → "being a part of", Eg floor in building. But can't ~~val~~ delete or create a floor out of nowhere!

→ It is an aggregate of a whole.

floor ⟵ ◇ building

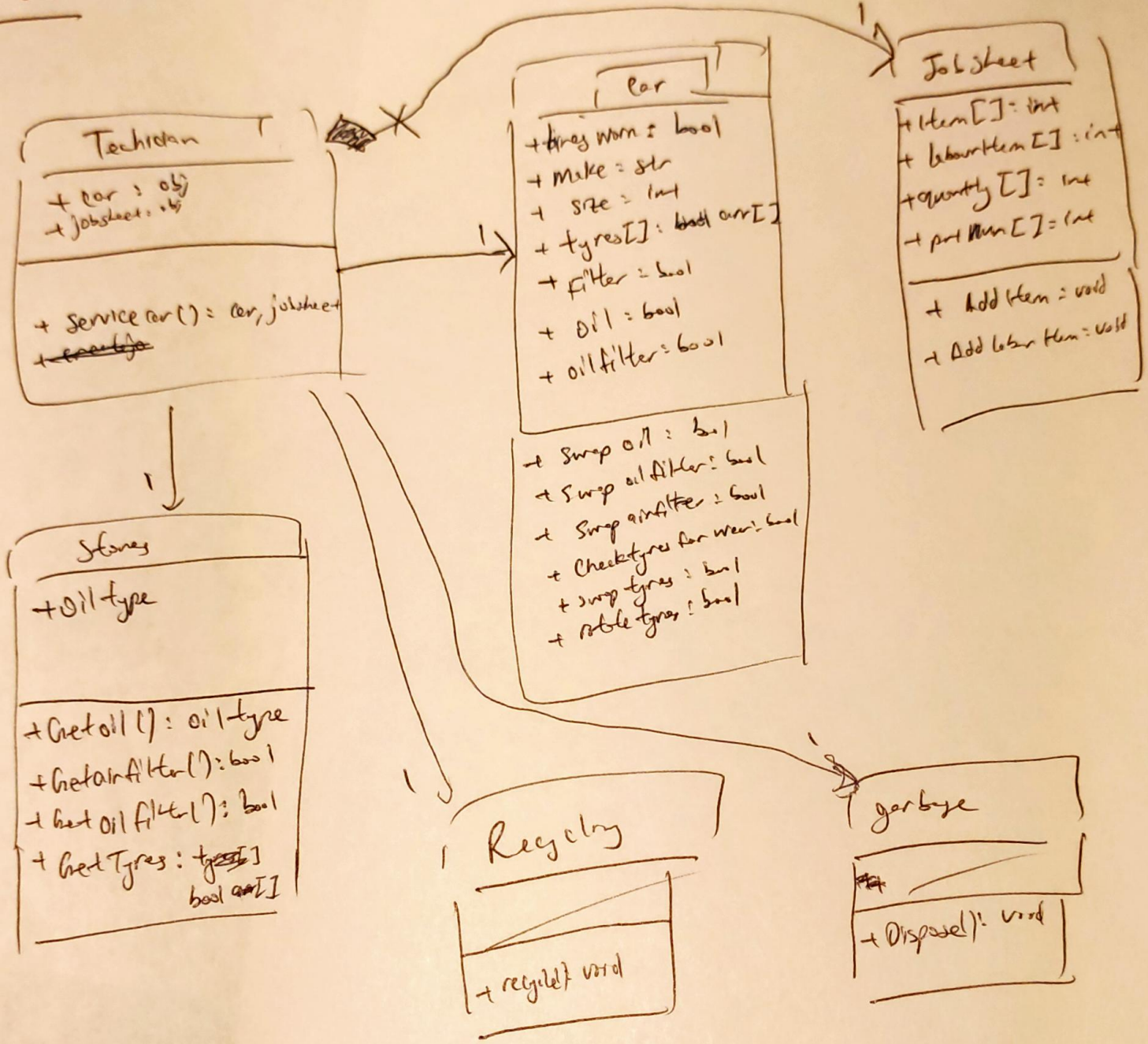
Composition → essentially a stronger aggregation where you can also create + delete, Eg ~~floor~~ baby within ~~building~~ human. Human on del + create as pieces.

Both are types of encapsulation

baby ⟵ ◆ human

~~baby ⟵ ◆ baby~~

Q 5B


Technician
+ car : obj
+ jobsheet : obj
+ service car() : car, jobsheet
Car
+ tires worn : bool
+ Make : str
+ size : int
+ tyres[] : bool arr[]
+ Filter : bool
+ Oil : bool
+ oilfilter : bool
+ Swap oil : bool
+ Swap oil filter : bool
+ Swap airfilter : bool
+ Checktyres for wear : bool
+ swap tyres : bool
+ rotate tyres : bool
Jobsheet
+ Item[] : int
+ labourItem[] : int
+ quantity[] : int
+ part Num[] : int
+ Add Item : void
+ Add labor Item : void
Storage
+ Oil type
+ Getoil() : oil type
+ Getairfilter() : bool
+ Get oil filter() : bool
+ Get Tyres : bool arr[]
Recycling
+ recycle() void
garbage
+ Disposal() : void