CS162

Discussion 7

EEVDF, I/O

10/30/24

Staff

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
		Midterm 2				
HW 4 Due			Project 2 Due			

EEVDF
Earliest Eligible Virtual Deadline First

Why not CFS?

Recall: Linux “Completely Fair Scheduler”

- One parameter: *niceness*
- Allows you to give a “slower” and “faster” physical runtime to tasks that corresponds to a unit of virtual runtime
- Tasks with higher priority will receive a larger real-life time slice on the CPU
  - Typically latency-sensitive or I/O bound!
- Proposal:
  - Add a new parameter: latency-nice patch
  - Ultimately depends on some heuristics to “guess” latency requirements => Difficult to predict

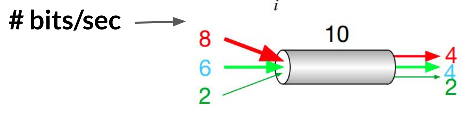
How do we allow tasks with urgent deadlines to skip the line without sacrificing “fairness”?

Fluid Flow System

- Idealized system where we serve multiple *flows* of data simultaneously.
  - Flow = continuous arrival of data or packets. 1 or more flows move through a link with a specified max **capacity.**
  - Each flow receives *min*(r_i, *f*) where r_i = *flow arrival rate* = rate at which flow *i* transmits data across the link.
  - *f* = *link fair rate* = fair share that should be distributed to each flow… unless, the flow requires *less* than fair rate.
  - Now, imagine each **flow** represents a **thread**. The **link** represents the **processor**.
- Each thread receives a **fair rate** of the processor **proportional to its weight** = how many bits of data it can send through the link at a time.
- **Basic system:**
  - Processor with capacity C serves N number of clients (threads). Each client initially deserves a **fair share *f*** = **C/N** = 10/3 = 3 ⅓
  - Client 3 only wants a share of 2 bits/sec→ recalculate **fair share *f*** to redistribute Client 3's share: (C-2) / (N-1) = 8/2 = 4.
- **Weighted system:**
  - fair share *f* = C/(w_1 + w_2 + w_3) = 10/5 = 2
  - Each flow should get *min*($w_{i,*}$f, r_i) = *min*(capacity of the link relative to its weight, its requested share)

Basic System:

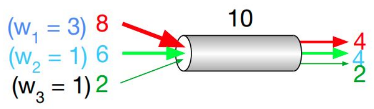
$$\sum_i \min(r_i, f) = C$$



f = 4:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

Weighted System:

$$\sum_i \min(r_i, f \times w_i) = C$$



f = 2:
min(8, 2*3) = 6
min(6, 2*1) = 2
min(2, 2*1) = 2

Generalized Processor Sharing

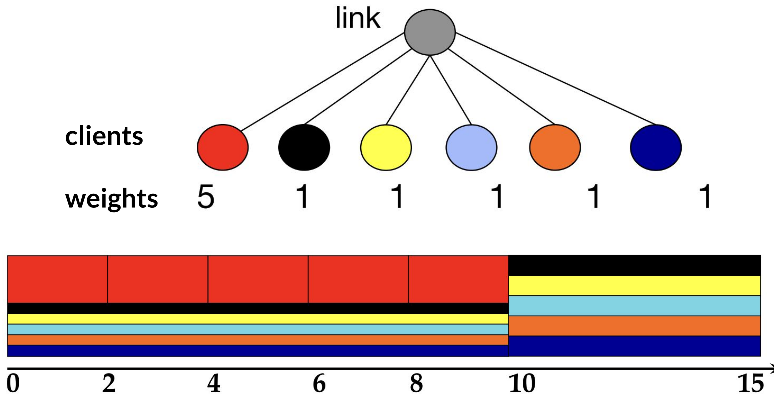
Generalized Processor Sharing

- Compare a *link* through which *flows* transmit *packets* to a *processor* that serves *threads* with different *requests*.
- *Processor* = shared resource across multiple clients (or *threads*).

Service time

- Service time is defined as time to access to a resource (i.e. CPU)
- A(t) = set of active clients at time *t*.
- Each thread should receive an **ideal service time** relative to its weighted share of the resource among all **active clients.**

GPS Example



CPU Capacity: 1 client/time unit. **Request length:** 1 time unit.

Red client: 5 requests. Other clients continuously submit requests.

Weighted share f_i for each client *i*

$$f_i(t) = \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j}.$$

Red client: f(t) = 0.5 for any t from [0, 10]

Other clients: f(t) = 0.1 for t from [0, 10]

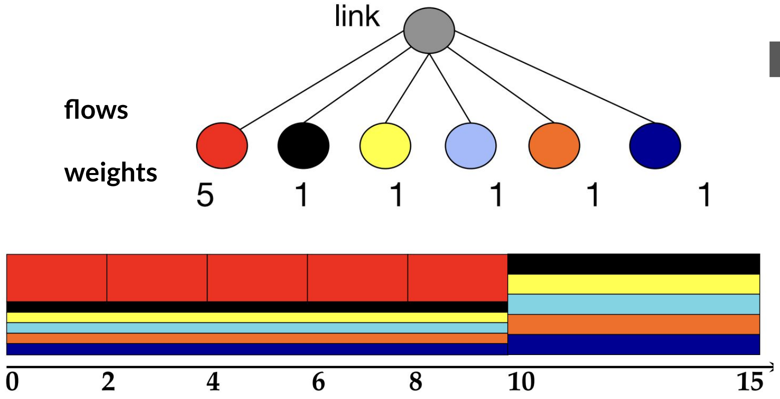
Ideal service time S_i for Client *i* from t_0 to t_1

Assume Thread i's weighted share (f_i) stays constant from t_0 to t_1

$$S_i\,(t_0,\, t_1) = f_i\,(t_0) \cdot (t_1 - t_0)$$

Fluid Flow → Generalized Processor Sharing

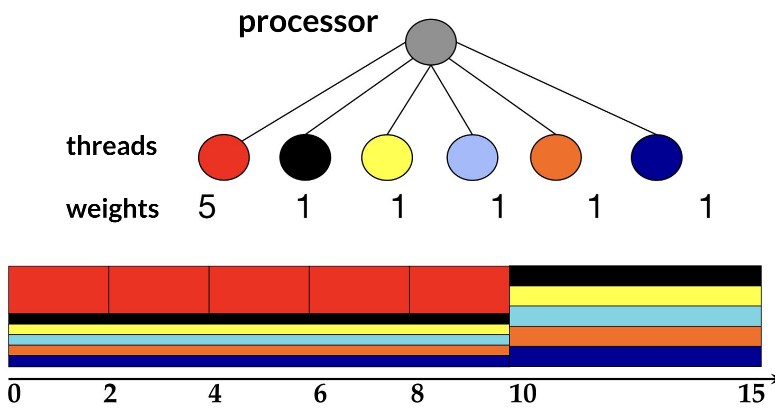
Fluid Flow System



Link Capacity: 1 bit/sec. **Packet size:** 1 bit.

Red flow: 5 packets. Other flows continuously receive packets.

Generalized Processor Sharing



CPU Capacity: 1 thread/time unit. **Request length:** 1 time unit.

Red thread: 5 requests. Other threads continuously submit requests.

Generalized Processor Sharing

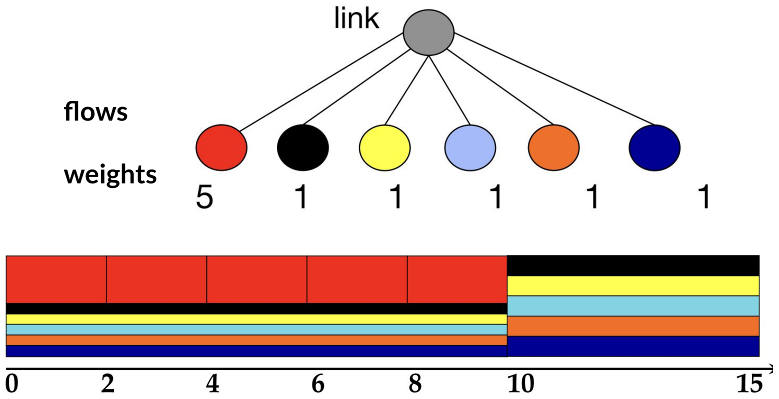
Generalized Processor Sharing

- Compare a *link* through which *flows* transmit *packets* to a *processor* that serves *threads* with different *requests*.
- We consider the link or processor a shared resource across multiple flows or threads.

Service time

- Service time is defined as time to access to a resource (i.e. CPU)
- A(t) = set of active clients (i.e. flows, threads) at time *t*.
- Each thread should receive an **ideal service time** relative to its weighted share of the resource among all **active clients.**

Fluid Flow System



Link Capacity: 1 bit/sec. **Packet size:** 1 bit.

Red flow: 5 packets. Other flows continuously receive packets.

Weighted share f_i for each client *i*

$$f_i(t) = \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j}.$$

Red flow: f(t) = 0.5 for any t from [0, 10]

Other flows: f(t) = 0.1 for t from [0, 10]

Ideal service time S_i for Client *i* from t_0 to t_1

Assume Client i's weighted share (f_i) stays constant from t_0 to t_1

$$S_i\,(t_0,\, t_1) \;=\; f_i\,(t_0) \,\cdot\, (t_1 - t_0)$$

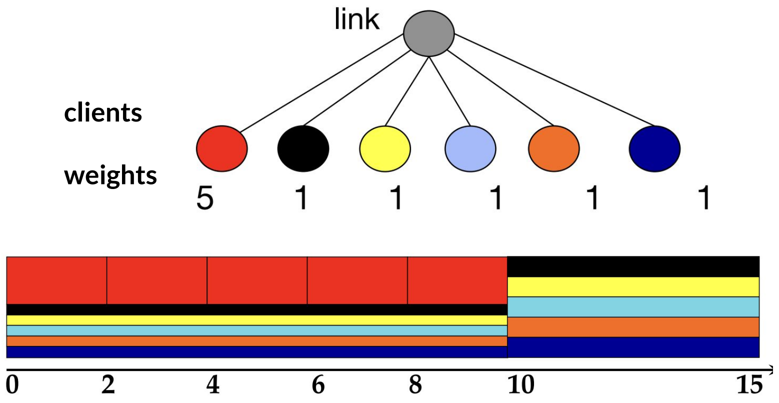
Concept Check

1. What is the ideal service time from t=0 to 1s for the red flow?

   What does this value represent?

2. What is the ideal service time from t=0 to 2s for any non-red flow?

Fluid Flow System



Link Capacity: 1 bit/sec. **Packet size:** 1 bit.

Red flow: 5 packets. Other flows continuously receive packets.

Ideal service time S_i for Client *i* from t_0 to t_1

Assume f_i stays constant from t_0 to t_1.

$$S_i\,(t_0,\,t_1)\;=\;f_i\,(t_0)\,\cdot\,(t_1-t_0)$$

Concept Check

1. What is the ideal service time from t=0 to 1s for the red flow?

   What does this value represent?

$S_i(0, 1) = 0.5 * (1-0) = 0.5s$

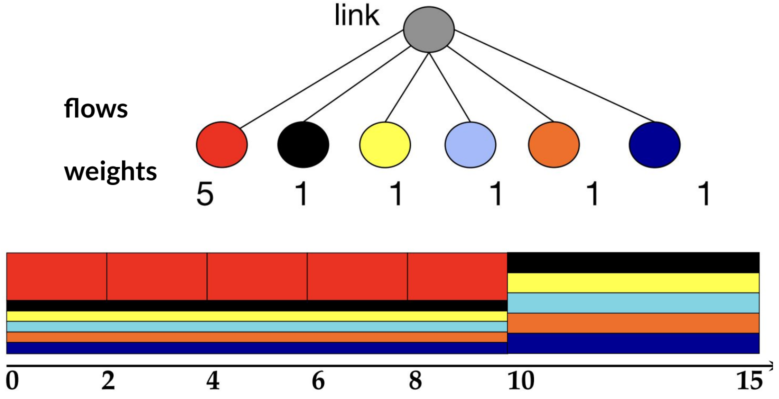
This means that the red flow should receive 1s of service time from t=0-2s.

From t=0-1s, the red flow is able to transmit 0.5 bits (i.e. ½ of a packet).

This is equivalent to having *full* access to the processor for 0.5s.

2. What is the ideal service time from t=0 to 1s for any non-red flow?

Fluid Flow System



Link Capacity: 1 bit/sec. **Packet size:** 1 bit.

Red flow: 5 packets. Other flows continuously receive packets.

Ideal service time S_i for Client *i* from t_0 to t_1

Assume f_i stays constant from t_0 to t_1.

$$S_i\left(t_0,\, t_1\right) = f_i\left(t_0\right) \cdot \left(t_1 - t_0\right)$$

Concept Check

1. What is the ideal service time from t=0 to 1s for the red flow?

   What does this value represent?

$S_i(0, 1) = 0.5 * (1-0) = 0.5s$

This means that the red flow should receive 1s of service time from t=0-2s.

From t=0-1s, the red flow is able to transmit 0.5 bits (i.e. ½ of a packet).

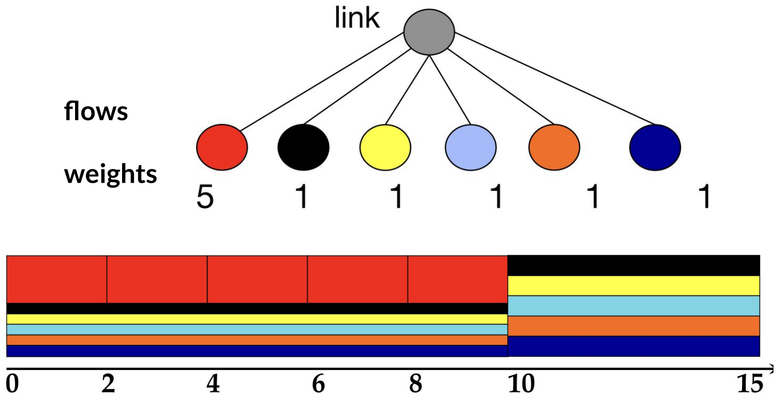
This is equivalent to having *full* access to the processor for 0.5s.

2. What is the ideal service time from t=0 to 1s for any non-red flow?

$S_i(0, 1) = 0.1 * (1-0) = 0.1s$

Same explanation as above.

Fluid Flow System



Link Capacity: 1 bit/sec. **Packet size:** 1 bit.

Red flow: 5 packets. Other flows continuously receive packets.

Ideal service time S_i for Client *i* from t_0 to t_1

Assume f_i stays constant from t_0 to t_1.

$$S_i(t_0, t_1) = f_i(t_0) \cdot (t_1 - t_0)$$

GPS and WFQ Example

Generalized Processor Sharing → Weighted Fair Queuing

- **Issue:** GPS assumes we can share the processor simultaneously among many threads.
- **Solution:** Consider a model where only 1 thread is given *complete* access of the shared resource (CPU) at any point in time.

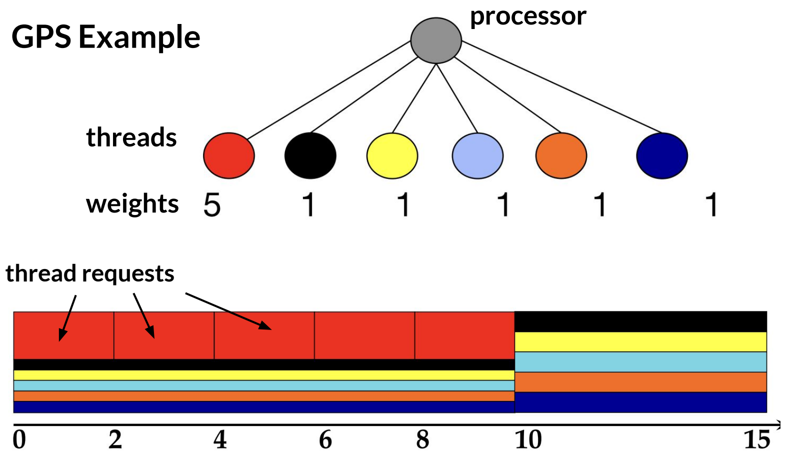
Scheduling

- Schedule threads in increments of a specified time quanta **q**.
- 1 round = each active thread had a chance to run, based on its weight.
- In each round, each client gets **w_i time units** to run on the CPU.
- Round duration = *sum(weights of active clients)*
- **WFQ:** Schedule clients based on when they **finish** in a fluid flow system.
  - No concept of eligible time… only looks at finishing time.

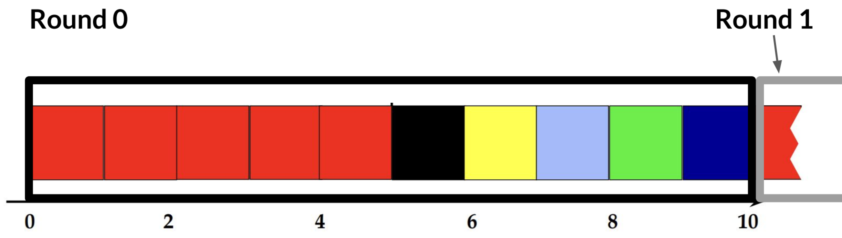
Issue with Weighted Fair Queueing

- WFQ doesn't spread out service of the red thread → needs to wait another 5 quanta before getting serviced. Red may become unresponsive :(
- **Solution:** In EEVDF, spread out the service of threads' requests by enforcing a time at which each request becomes eligible.

GPS Example



Weighted Fair Queuing



Red thread has 50% of total weight → gets 50% of the CPU share per round.

EEVDF vs. Generalized Processor Sharing

Modeling Generalized Processor Sharing

- Threads can make multiple *requests*, which should be scheduled in the order in which they were submitted.
- Each request (analogous to a packet) is **eligible** in a fluid flow system after the previous thread's request finished executing (after prev. deadline)
- Recall that when scheduling threads, *packets* are equivalent to *requests* for different threads.

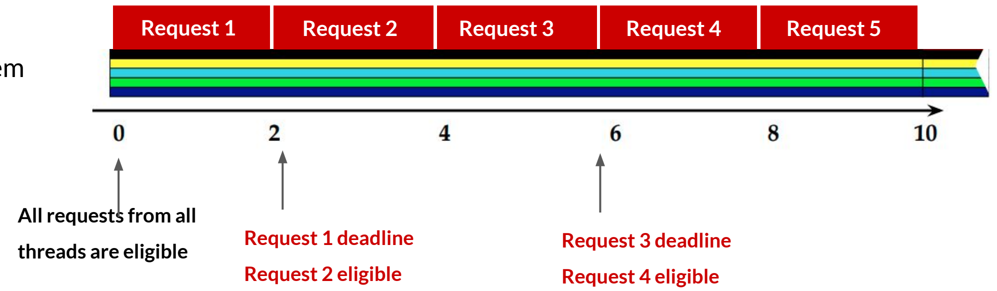
Eligible Time and Deadline

- **Eligible time** = time when packet/request can be scheduled.
- **Deadline** = time by which packet/request should be served, ideally.

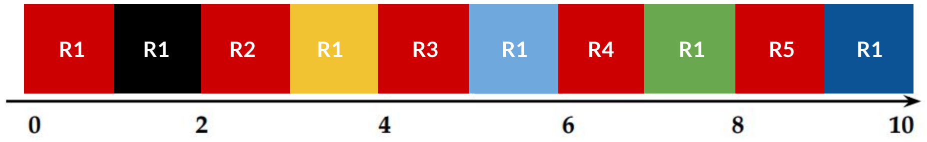
Scheduling Decisions

- At each time step, only consider packets that are **eligible** (i.e. **e** < **t**, where e = eligible time, t = current time)
- Choose the packet/request with the **earliest deadline d.**

Generalized Processor Sharing



(Round 0)



EEVDF: Service time and lag

Service Time

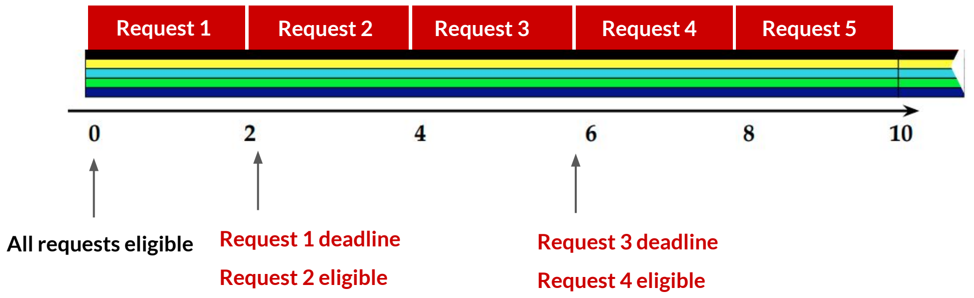
- **Real service time** = service time client *i* **actually** receives in a non-ideal system in a given time interval.
- **Ideal service time** = service time client *i* **should** receive based on an ideal system (i.e. client's weighted share * time interval).

Lag: ideal service time **-** real service time.

Positive lag: Underallocated amount of promised service time.

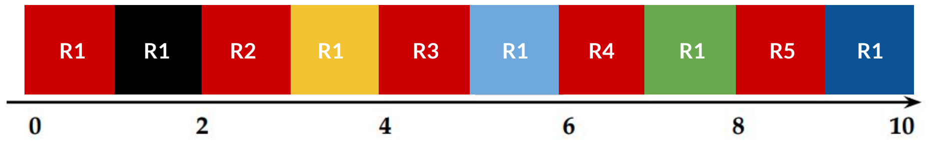
Negative lag: Overallocated amount of promised service time.

Fluid Flow System



EEVDF

Request length = ideal service time = 1 time unit.



Virtual Time

Rate of Virtual Time

- Rate of virtual time is *inversely proportional* to the sum of weights *of active clients.*
- Virtual time moves slower with more active clients.
  - Imagine a round robin scheduler with n threads.
  - If another thread enters the system, the *round* takes more time, but the *order* in which threads finish *does not change.*

Virtual Time

- V(t) = virtual time at time *t* is the *index of the round at time t.*
- Recall, round duration (in terms of quanta) = sum of the weights of active threads.

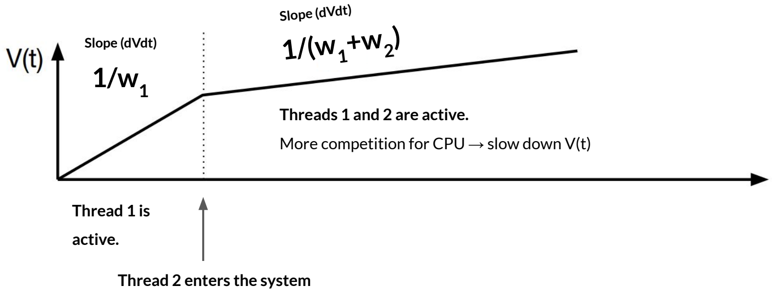
$$V(t_1) = \int_0^{t_1} \frac{1}{\sum_{j \in A(t)} w_j} \cdot dt$$

- Consider a time interval (t_1 , t) where the active set of threads **stays constant.**

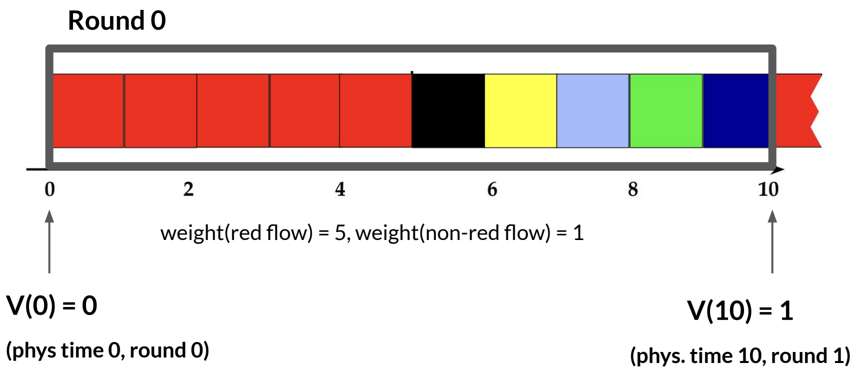
$$V(t) - V(t_1) = \frac{1}{\sum_{j \in A(t)} w_j} \cdot (t - t_1)$$

- In EEVDF, we make scheduling decisions based on **virtual time** or the **order** in which threads should start and finish.

Rate of Virtual Time



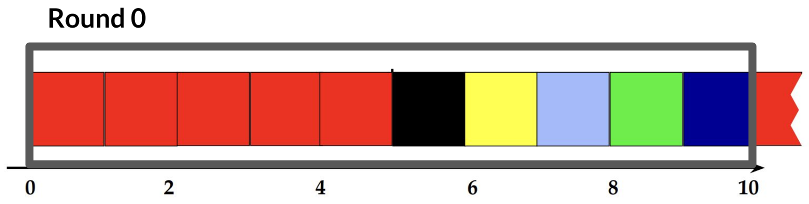
Weighted Fair Queuing



Concept Check

1. In this example of weighted fair queueing, what is the **virtual time** at physical time t=5 in this example?

Assume the set of active clients include all flows and stays constant from t=0 to 10.



weight(red flow) = 5, weight(non-red flow) = 1

2. Now, assume that the **yellow client** enters the system at t=6 instead of t=0, while all others are already present.

   What is the **virtual time** at physical time t=5?

Concept Check

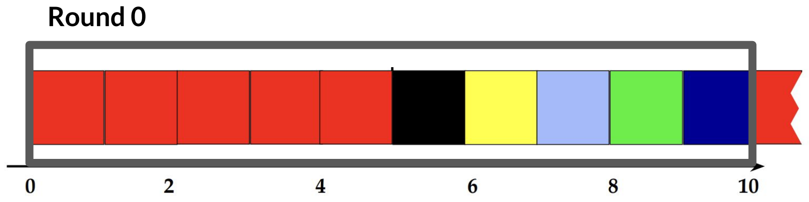
1. In this example of weighted fair queueing, what is the **virtual time** at physical time t=5 in this example?

Assume the set of active clients include all flows and stays constant from t=0 to 10.

V(t) = 1/(sum of weights of active clients) * t

V(t) = 1/10 * t

V(t) = 1/10 * 5 = 0.5



weight(red flow) = 5, weight(non-red flow) = 1

2. Now, assume that the **yellow client** enters the system at t=6 instead of t=0, while all others are already present.

   What is the **virtual time** at physical time t=5?

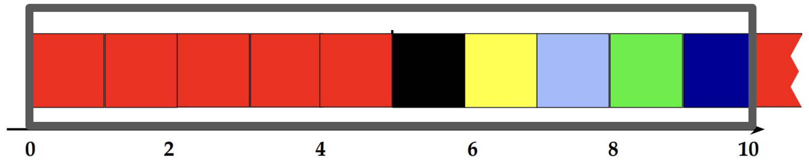
3. From the scenario above, what is the virtual time at time t=7?

Concept Check

1. In this example of weighted fair queueing, what is the **virtual time** at physical time t=5 in this example?

Assume the set of active clients include all flows and stays constant from t=0 to 10.

Round 0



weight(red flow) = 5, weight(non-red flow) = 1

V(t) = 1/(sum of weights of active clients) * t

V(t) = 1/10 * t

V(t) = 1/10 * 5 = 0.5

2. Now, assume that the **yellow client** enters the system at t=6, while all others are already present.

   What is the **virtual time** at physical time t=5?

V(t) = 1/(sum of weights of active clients) * t

V(t) = 1/9* t

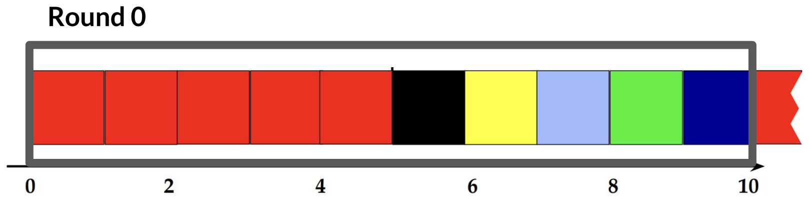
V(t) = 1/9 * 5 = 0.55

3. From the scenario above, what is the virtual time at time t=7?

Concept Check

1. In this example of weighted fair queueing, what is the **virtual time** at physical time t=5 in this example?

Assume the set of active clients include all flows and stays constant from t=0 to 10.



weight(red flow) = 5, weight(non-red flow) = 1

V(t) = 1/(sum of weights of active clients) * t

V(t) = 1/10 * t

V(t) = 1/10 * 5 = 0.5

2. Now, assume that the **yellow client** enters the system at t=6, while all others are already present.

   What is the **virtual time** at physical time t=5?

V(t) = 1/(sum of weights of active clients) * t

V(t) = 1/9* t

V(t) = 1/9 * 5 = 0.55

3. From the scenario above, what is the virtual time at time t=7?

V(6) = 0.67, and virtual time from t=6 to 7 => 1/10 * 1s = 0.1

V(7) = V(6) + (V(7) - V(6)) = 0.67 + 0.1 = 0.77

Note that you cannot just take **1/10 * 7s** because the **yellow client** only enters the system at t=6.

EEVDF: The Easy Version

Every quantum I want to hand out, I want to pick a thread to run like this:

1. Pick the most “urgent” task with the earliest deadline
2. Let it run on the processor for the whole quantum
   a. If it finishes early, we reward it.
      i. => Allow it to be scheduled earlier.
      ii. Earlier deadline as a result
   b. If it spends too much time, we punish it.
      i. => Force it to be scheduled later
      ii. Later deadline as a result
3. Repeat.

We permit shorter tasks to “skip the line”!

EEVDF: Definitions

- **Request interval *R*:**
  - The requested time slice for thread **i**
- **Virtual eligible time:** earliest virtual time at which a thread is *eligible* to be scheduled.
  - Naturally, the first eligible time (V_e^0) is when the thread first arrives.
- **Virtual deadline:** earliest virtual time by which a thread should be fully serviced.
  - Current virtual eligible time + client's request interval scaled down by the client's fair share rate ($\mathbf{f_i}$)
  - $\mathbf{V_d = V_e + R/f_i}$

Recall that in physical reality, we might have threads receiving more or less than the requested length **R.**

- **Lag:** ideal service time **-** real service time.
  - Say thread **i** receives some **u** timeslice as opposed to ***R*.**
  - Alternate expression => $\mathbf{(R - u)/w_i}$.
  - Abstract this away as the difference between the "ideal" and "reality"

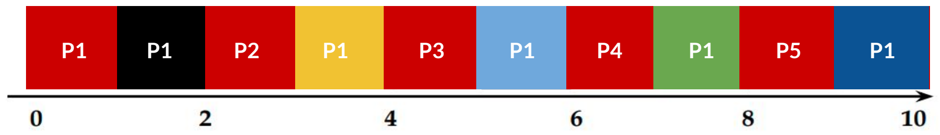
EEVDF: Algorithm

- For every quantum **q**:
  - Get all **eligible** requests (i.e. eligible time ≤ current time).
  - Select the request with the **earliest virtual deadline.**
- Compute next states based on lag.
  - Your new virtual eligible time: $\mathbf{V_e^{i+1} = V_d - lag}$
  - Your new virtual deadline: $\mathbf{V_d^{i+1} = V_e^{i+1} + R/f_i}$

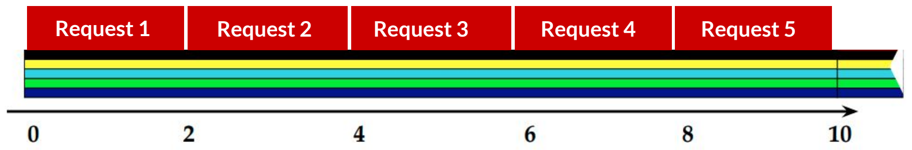
    Rinse and repeat!
  - In words: Your next eligible time gets compensated by the “lag” you experienced.
    - Positive lag pulls the next eligible time closer. Negative lag postpones it further away.
- Observation:
  - Flows with positive lag are always eligible.

Concept Check

EEVDF



Fluid Flow System

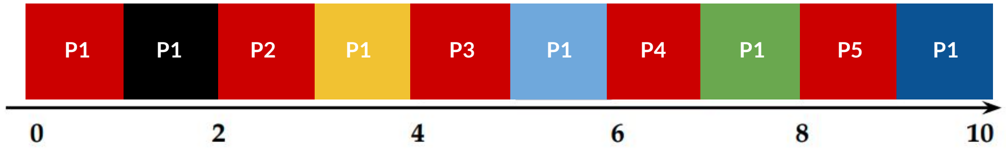


1. From t=0 to 2s, why must red packet 1 be scheduled before any non-red packet? Why does this also apply for t=4 to 6s for red packet 3?

2. Now, assume we scheduled the blue packet before red packet 5. What is the lag from t=8-9s for the red flow?

Concept Check

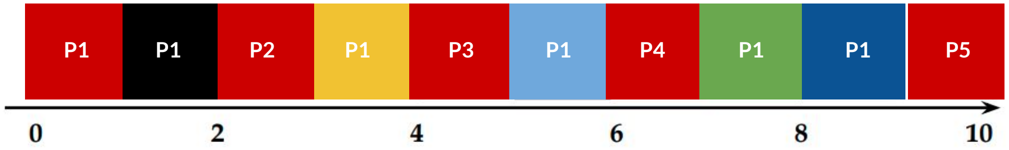
1. From t=0 to 2s, why must red packet 1 be scheduled before any non-red packet? Why does this also apply for t=4 to 6s?



From t=0 to 2s, red packet 1 and all non-red packets are eligible, *but* red packet 1 has the **earliest deadline.** Thus, red packet 1 must be scheduled first.

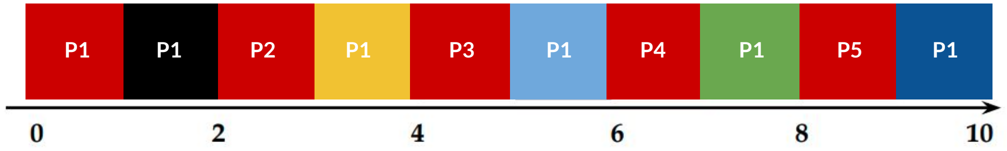
The same applies for t=4 to 6s because red packet 3 has a deadline of **t=6s**, whereas all non-red packets have a deadline of **t=10s.**

2. Now, assume we scheduled the blue packet at t=8 and the last red packet at t=9. What is the lag from t=8-9s for the red flow?



Concept Check

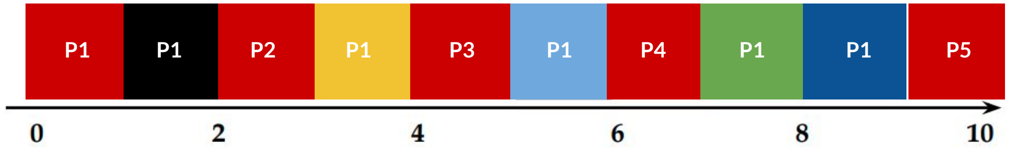
1. From t=0 to 2s, why must red packet 1 be scheduled before any non-red packet? Why does this also apply for t=4 to 6s?



From t=0 to 2s, red packet 1 and all non-red packets are eligible, *but* red packet 1 has the **earliest deadline.** Thus, red packet 1 must be scheduled first.

The same applies for t=4 to 6s because red packet 3 has a deadline of **t=6s**, whereas all non-red packets have a deadline of **t=10s.**

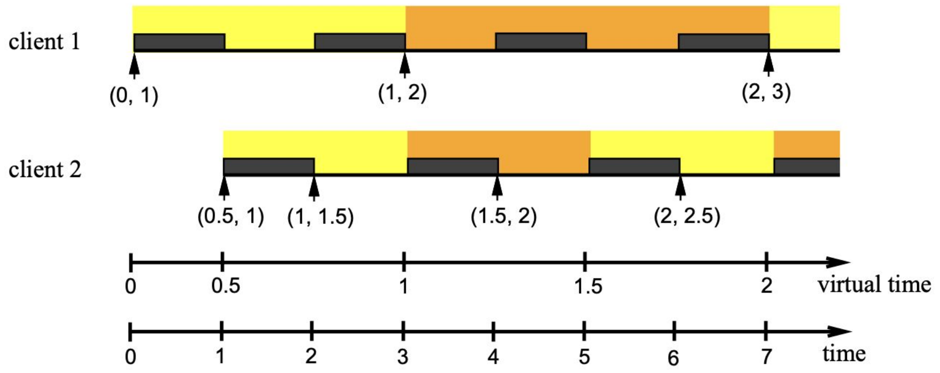
2. Now, assume we scheduled the blue packet at t=8 and red packet 5 at t=9. What is the lag from t=8-9s for the red flow?



Ideal service time = 0.5*1s = 0.5s, Received service time = 0s → lag = 0.5s - 0s = 0.5s → **positive lag** (red flow “owed” CPU time)

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=v_e+R/w_1 = 1	—



<u>Client 1</u>
w_1=2
r_1=2 quanta

<u>Client 2</u>
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Calculate **Thread i's** ve, vd for their **kth request.**

t_0^i = time that request was initiated.

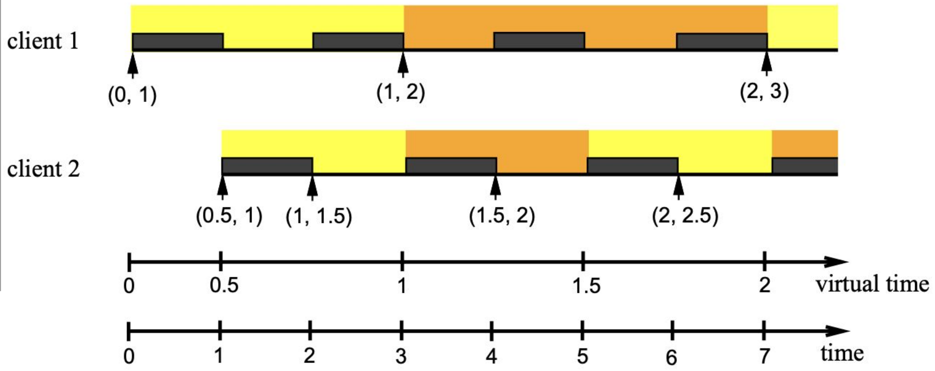
Eq. 1 $ve^{(1)} = V(t_0^i),$

Eq. 2 $vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$

Eq. 3 $ve^{(k+1)} = vd^{(k)}.$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0, =v_d=1	—
1	0.5	C1, C2	**0.25**	v_e= 0, v_d= 1	t_0^i = 1 v_e = v(1) = 1/w_1 * 1 = ½ , v_d= v_e + r/w_2



Recall, V(t) = virtual time at physical time t

Physical time of initial request for Client 2 = t_0^2 = 1

V_e = V(t_0^2) = V(1) = 1/w_1*(t-0) = 1/w_1*1 = 0.5

V_d = V_e + r/w_2 = 0.5 + 0.5 = 1

Next(V_e) = V_d = 1 (assuming no lag).

Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

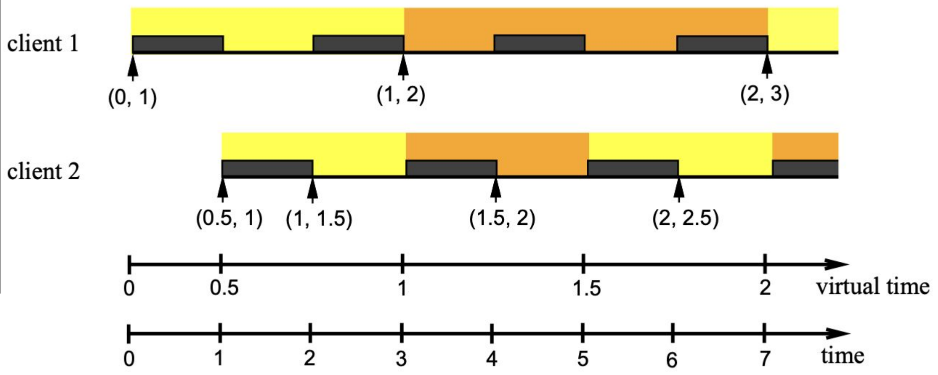
Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	C1	**0.5**	v_e=0, =v_d=1	—
1	0.5	C1, C2	**0.25**	v_e= 0, v_d= 1	$t_0^i = 1$ $v_e = v(1) = 1/w_1 * 1 = ½$, $v_d = v_e + r/w_2$

New active thread → rate of virtual time *from t=1 onwards* changes.

dVdt = $1/(w_1 + w_2) = 0.25$



Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

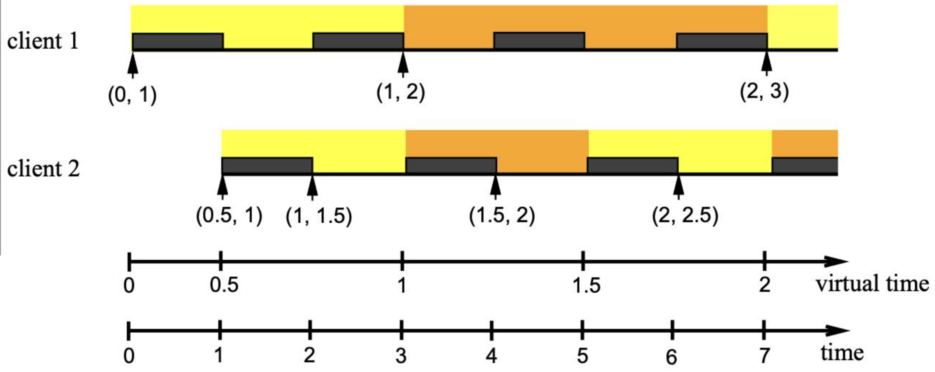
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, **C2**	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1



1. Both clients are *eligible* since the current virtual time >= their respective **virtual eligible times**
2. Both clients share the *same* **virtual deadline**

In this case, we arbitrarily tie-break (let's say client 2 runs for the next quanta in this example

<u>Client 1</u>
w_1=2
r_1=2 quanta

<u>Client 2</u>
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

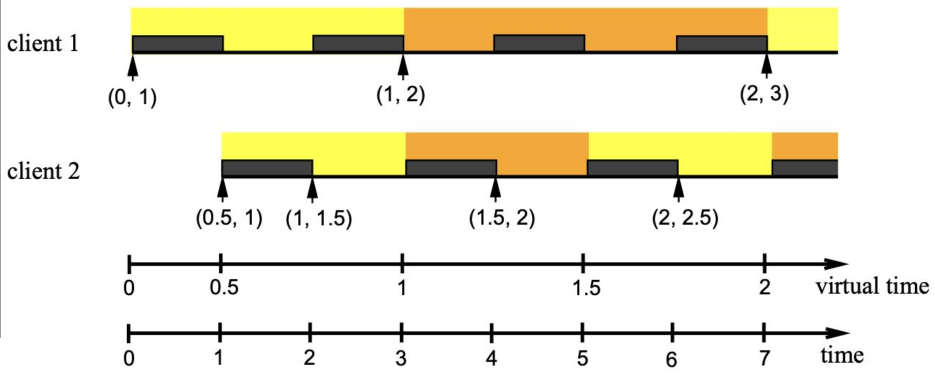
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	**dV(t) / dt**	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, **C2**	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	**C1**, ~~C2~~	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5



Client 1
Client 1 has not finished its initial request → V_e , V_d do not change.

Client 2
Finished request → recalculate V_e, V_d. No lag → V_e = prev V_d
$V_d = V_e + r/w = 1.5$

<u>Client 1</u>
w_1=2
r_1=2 quanta

<u>Client 2</u>
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

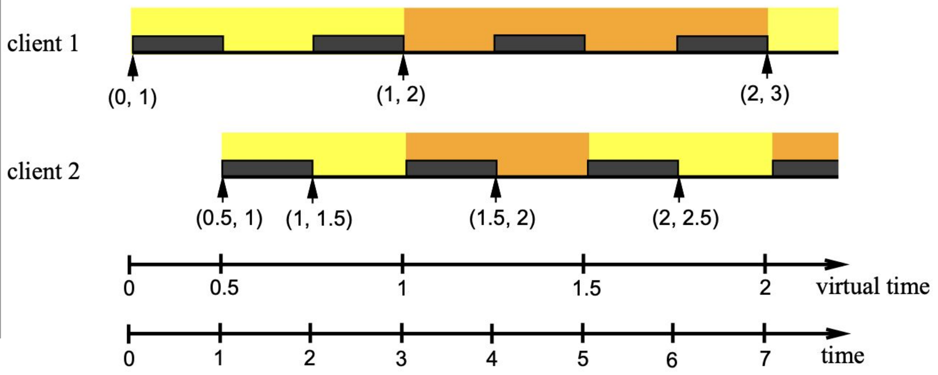
Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	C1	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, C2	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	C1, ~~C2~~	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5

Client 2's request has been fulfilled. So we re-evaluate client 2's new **virtual eligible time** and **virtual deadline**.

Trivially, only Client 1 can be scheduled now so give this quanta to Client 1



Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

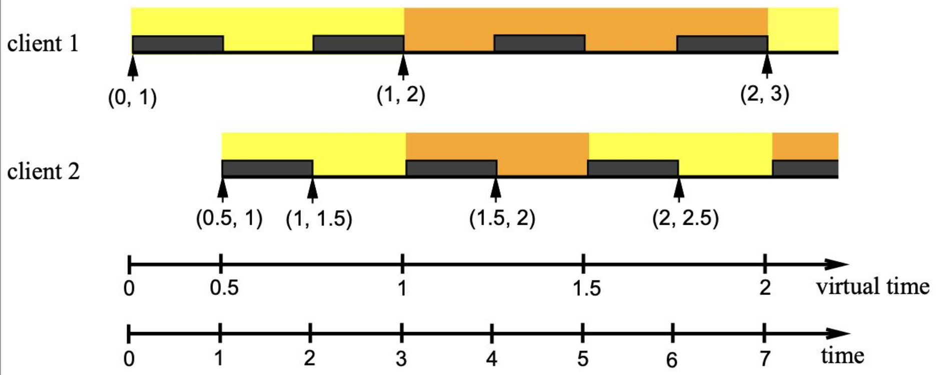
Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	**dV(t) / dt**	Client 1 (C1)	Client 2 (C2)
0	0	C1	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, C2	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	C1, C2	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5
3	1	C1, C2	**0.25**	v_e=1,v_d=2	v_e=1, v_d=1.5

Client 1 has fulfilled its request.
Re-evaluate its new **virtual eligible time** and **virtual deadline**.

Since Client 2 has an *earlier* **virtual deadline**, we schedule Client 2 next.



Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

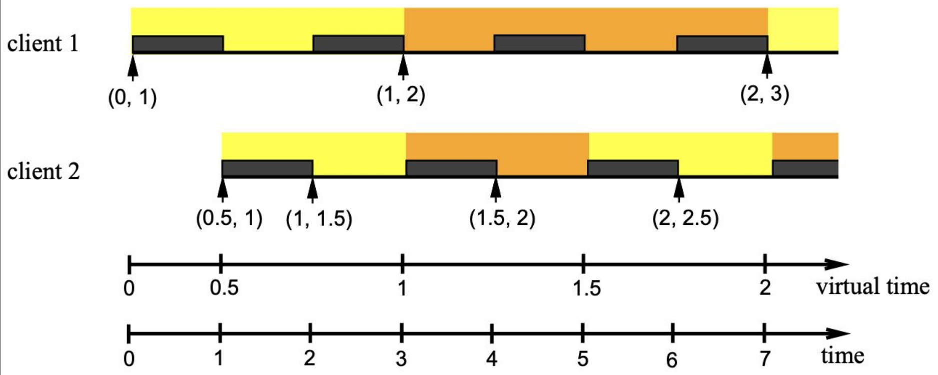
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	dV(t) / dt	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, **C2**	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	**C1**, C2	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5
3	1	C1, **C2**	**0.25**	v_e=1,v_d=2	v_e=1, v_d=1.5
4	1.25	**C1**, C2	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2



Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Eq. 1 $$ve^{(1)} = V(t_0^i),$$

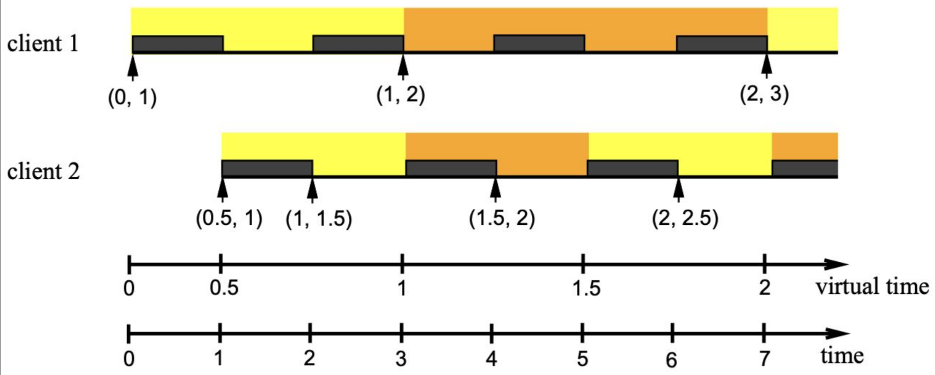
Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	C1	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, C2	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	C1, C2	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5
3	1	C1, C2	**0.25**	v_e=1,v_d=2	v_e=1, v_d=1.5
4	1.25	C1, C2	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2
5	1.5	C1,C2	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2

Both clients are eligible again. Both have the *same* **virtual deadline**.
Tie-break through arbitrary manner. We choose C2 here again.



Client 1
w_1=2
r_1=2 quanta

Client 2
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

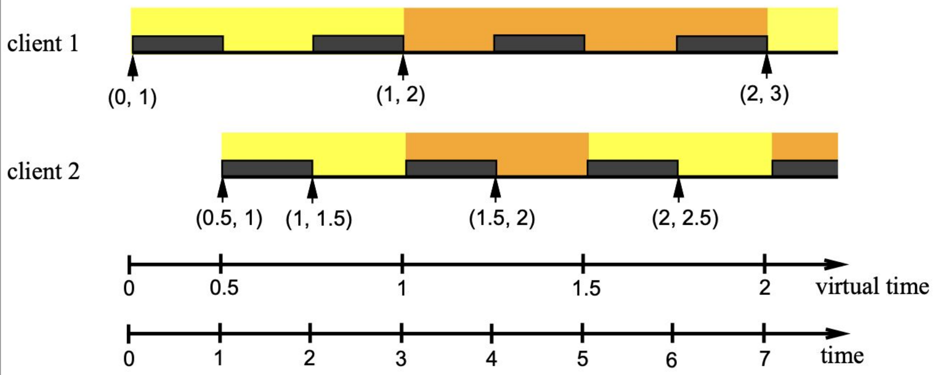
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, **C2**	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	**C1**, C2	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5
3	1	C1, **C2**	**0.25**	v_e=1,v_d=2	v_e=1, v_d=1.5
4	1.25	**C1**, C2	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2
5	1.50	C1,**C2**	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2
6	1.75	**C1**, C2	**0.25**	v_e=1,v_d=2	v_e=2, v_d=2.5



<u>Client 1</u>
w_1=2
r_1=2 quanta

<u>Client 2</u>
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

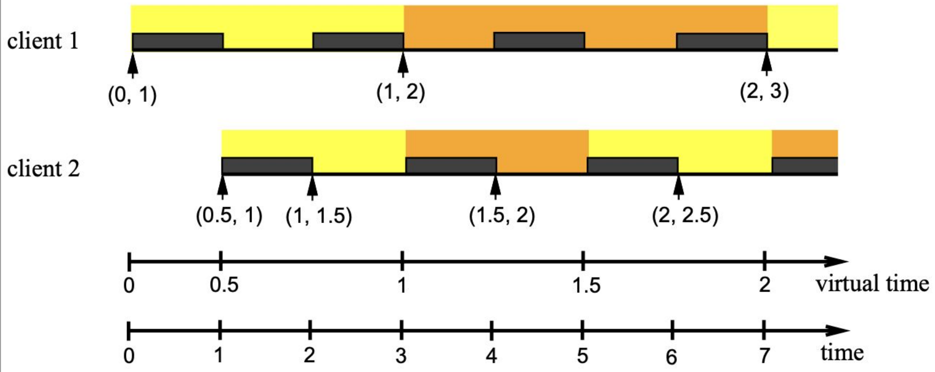
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

Simple EEVDF Example

t	V(t)	Active Threads	**dV(t) / dt**	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	v_e=0 (trivial), v_d=1	—
1	0.5	C1, **C2**	**0.25**	v_e= 0, v_d=1	v_e=0.5, v_d=1
2	0.75	**C1**, C2	**0.25**	v_e=0, v_d=1	v_e=1, v_d=1.5
3	1	C1, **C2**	**0.25**	v_e=1,v_d=2	v_e=1, v_d=1.5
4	1.25	**C1**, C2	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2
5	1.5	C1,**C2**	**0.25**	v_e=1,v_d=2	v_e=1.5, v_d=2
6	1.75	**C1**, C2	**0.25**	v_e=1,v_d=2	v_e=2, v_d=2.5
7	2	C1,**C2**	**0.25**	v_e=2,v_d=3	v_e=2, v_d=2.5



<u>Client 1</u>
w_1=2
r_1=2 quanta

<u>Client 2</u>
w_2= 2
r_2= 1 quanta

q = 1s

Recurrence Relation to calculate *(ve, vd)*

Eq. 1 $$ve^{(1)} = V(t_0^i),$$

Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

Eq. 3 $$ve^{(k+1)} = vd^{(k)}.$$

I/O

I/O

Access patterns

- **Character devices** access data as a character stream (i.e. data not addressable).
- **Block devices** access data in fixed-sized blocks (i.e. data is addressable).
- **Network devices** have a separate interface for networking purposes.

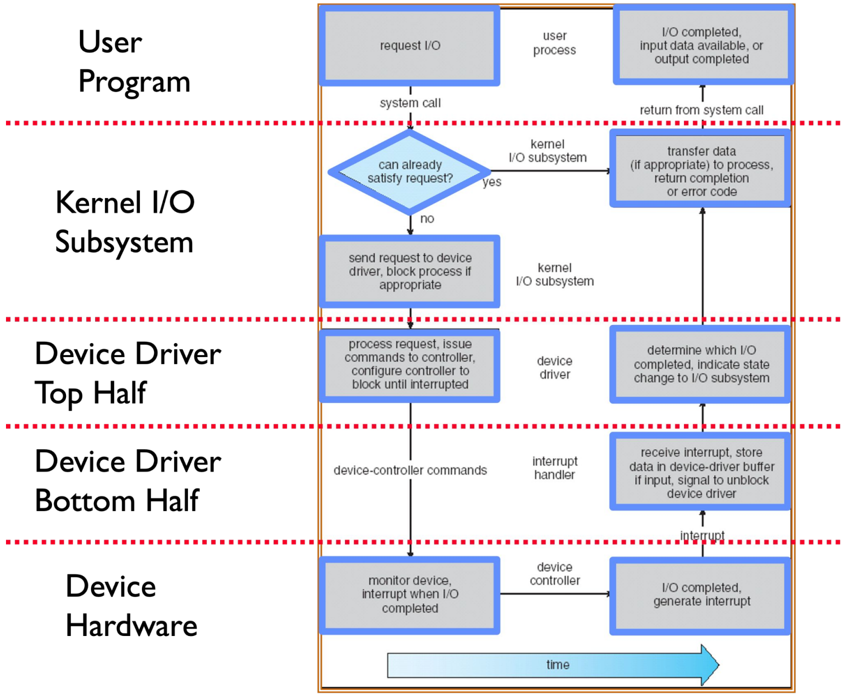
Access timing

- **Synchronous**/**blocking** interfaces wait until an I/O request is fulfilled.
- **Non-blocking** interfaces return quickly from a request without waiting.
- **Asynchronous** interfaces allow for other processing to continue without waiting.
- Asynchronous is not the same as non-blocking!

I/O

Device drivers connect high-level abstractions implemented by the OS and hardware specific details of I/O devices.

- Allows kernel to use a standard interface for I/O devices.
- **Top half** is used by the kernel to start I/O operations.
- **Bottom half** is used to service interrupts from the device.
- In Linux, top and bottom are flipped!


User Program
Kernel I/O Subsystem
Device Driver Top Half
Device Driver Bottom Half
Device Hardware
request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Device Access

Processors talk to I/O devices through **device controllers**

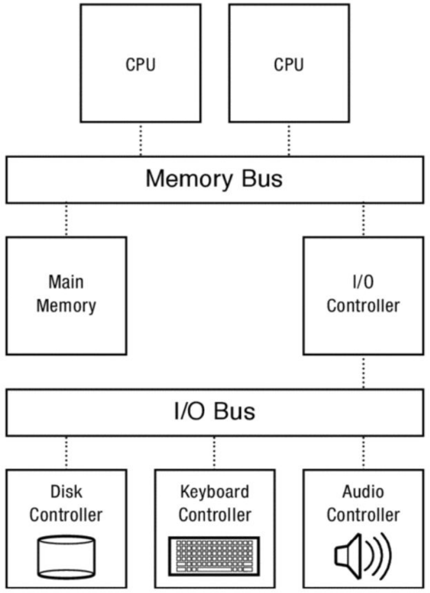
- Contain a set of registers that can be read from/written to.

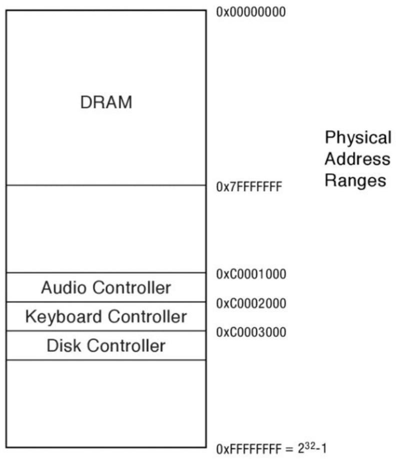
Programmed I/O (PIO) involves the processor in every byte transfer.

- **Port-mapped I/O (PMIO)**
  - Uses special memory instruction (e.g. in, out) → complicates CPU logic with special instructions.
  - Uses distinct memory address space from physical memory → useful on older/smaller devices with limited physical address space.
- **Memory-mapped I/O (MMIO)**
  - Uses standard memory instructions (e.g. load, store) → simplifies CPU logic by putting the responsibility on the device controller
  - Shares memory address with physical memory.

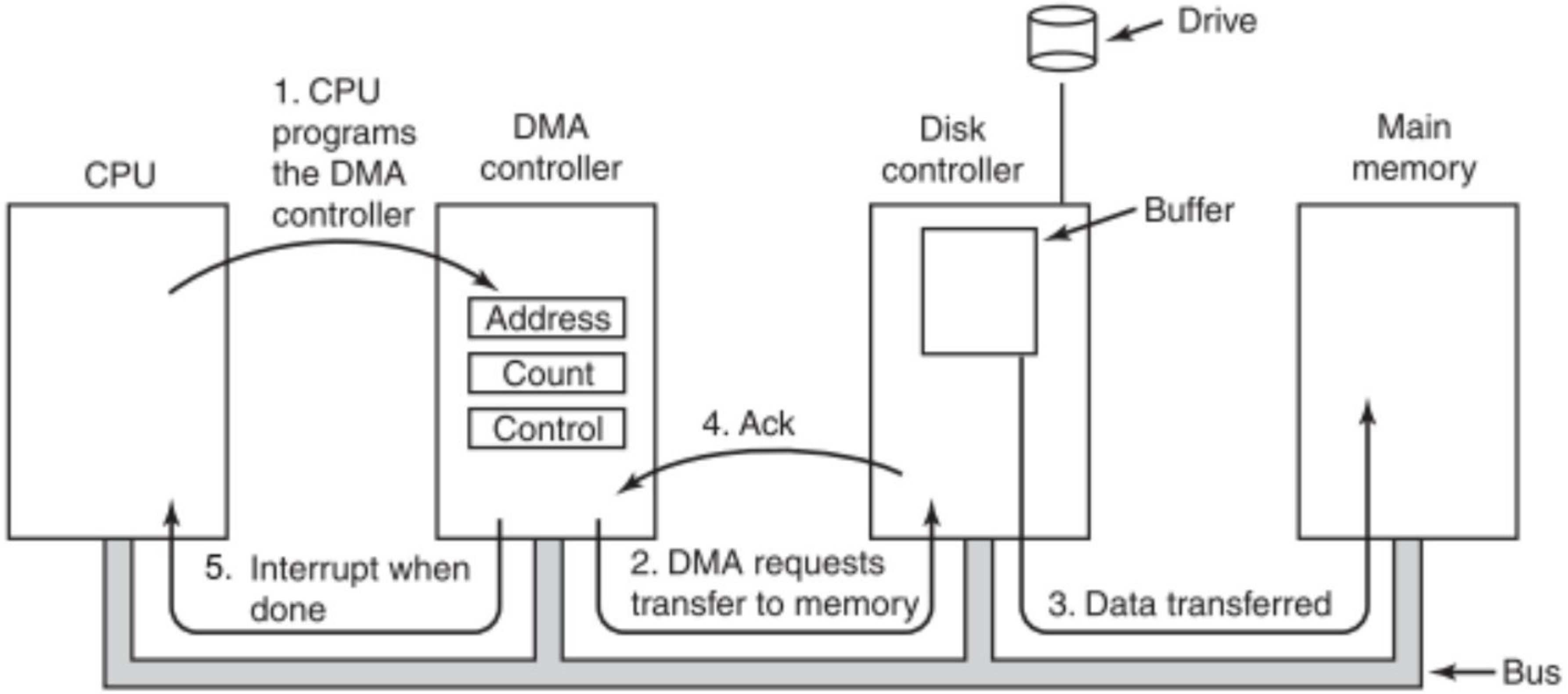
Direct memory access (DMA) allows device to write to main memory without CPU intervention.

- Uses memory addresses within physical memory region, not just the physical address space.





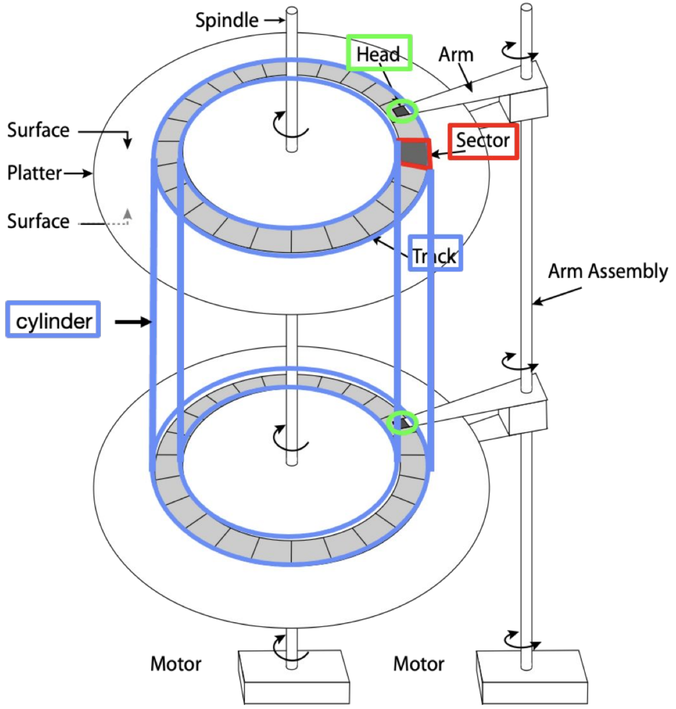
DMA Walkthrough



Storage Devices (HDD)

Magnetic disks use magnetization to read/write data.

- **Platter** is a single thin round plate that stores information.
- Each platter spins on a **spindle**, and each platter has two **surfaces**.
- Data is read and written with the **head**.
- Each head attaches to an **arm**, and each **arm** attaches to the **arm assembly.**
- Data is stored in fixed size units called **sectors**, the minimum unit of transfer.
- Circle of sectors make up a **track**.
- Track length varies from inside to outside of the disk.
- Lots of moving parts → physical repair issues.
- Read/write times slower compared to flash.
- Much cheaper than flash.



Storage Devices

Flash memory uses electrical circuits (e.g. NOR, NAND) for persistent storage.

- Focus on NAND flash which reads/writes in fixed size units called **pages** (typically 2 KB to 4 KB).
- Much less moving parts → less prone to issues.
- Read/write speeds much faster.
- Writing to a cell requires erasing it first, in large fixed size units called **erasure blocks**.
- Each page has a finite lifetime, can only be erased and rewritten a fixed number of times (e.g. 10K)

Flash translation layer (FTL) maps logical flash pages to different physical pages.

- Allows device to relocate data without the OS knowing/worrying.
- Can write data by writing to a new location, updating mapping in FTL, then erasing old page in the background.
- Ensure all physical pages are used equally using **wear levelling**.

Storage Devices

Time to access data is composed of

- **Queueing time** refers to how long a data request spends in the OS queue before being passed to the device controller.
- **Controller time** refers to how long the device controller spends processing the request.
- **Access time** refers to how long it takes to access data from the device.

Magnetic disks' access time involves

- **Seek time**: move the arm over the desired track.
- **Rotation time**: wait for target sector to rotate under head.
- **Transfer time**: transfer data to/from buffer for read/write
- Aim to minimize seek and rotation since transfer is fixed.

Use disk intelligence to improve performance.

- **Track skewing** staggers logical sectors of the same number on each track by the time it takes to move across one track.
- **Sector sparing** remaps bad sectors to spare sectors.
- Use **buffer memory** to store sectors read by disk head but not requested by the OS

Concept check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

3. When using SSDs, which between reading or writing data is complex and slow?

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

Concept check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

   Yes. Only with non-blocking IO can you have concurrency without multiple threads.

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

3. When using SSDs, which between reading or writing data is complex and slow?

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

Concept check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

   Yes. Only with non-blocking IO can you have concurrency without multiple threads.

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

   No. It is more efficient to poll, since the CPU will get overwhelmed with interrupts.

3. When using SSDs, which between reading or writing data is complex and slow?

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

Concept check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

   Yes. Only with non-blocking IO can you have concurrency without multiple threads.

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

   No. It is more efficient to poll, since the CPU will get overwhelmed with interrupts.

3. When using SSDs, which between reading or writing data is complex and slow?

   SSDs have complex and slower writes because their memory can't easily be mutated.

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

Concept check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

   Yes. Only with non-blocking IO can you have concurrency without multiple threads.

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

   No. It is more efficient to poll, since the CPU will get overwhelmed with interrupts.

3. When using SSDs, which between reading or writing data is complex and slow?

   SSDs have complex and slower writes because their memory can't easily be mutated.

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

   Transferring large amounts of data which take a long time to not involve the CPU.

Concept Check

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
    a. If you had to choose where to lay out these “back-up” sectors on disk - where would you put them? Why?
    b. How do you think that the disk controller can check whether a sector has gone bad?
    c. Can you think of any drawbacks of hiding errors like this from the OS?
6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?

Concept Check

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
    a. If you had to choose where to lay out these "back-up" sectors on disk - where would you put them? Why?
    Should spread them out evenly, so when you replace an arbitrary sector your find one that is close by.
    b. How do you think that the disk controller can check whether a sector has gone bad?
    c. Can you think of any drawbacks of hiding errors like this from the OS?
6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?

Concept Check

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
   a. If you had to choose where to lay out these “back-up” sectors on disk - where would you put them? Why?
      Should spread them out evenly, so when you replace an arbitrary sector your find one that is close by.
   b. How do you think that the disk controller can check whether a sector has gone bad?
      Using a checksum, this can be efficiently checked in hardware during disk access.
   c. Can you think of any drawbacks of hiding errors like this from the OS?

6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?

Concept Check

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
    a. If you had to choose where to lay out these "back-up" sectors on disk - where would you put them? Why?
    Should spread them out evenly, so when you replace an arbitrary sector your find one that is close by.
    b. How do you think that the disk controller can check whether a sector has gone bad?
    Using a checksum, this can be efficiently checked in hardware during disk access.
    c. Can you think of any drawbacks of hiding errors like this from the OS?
    Excessive sector failures are warning signs that a disk is beginning to fail.

6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?

Concept Check

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
   a. If you had to choose where to lay out these "back-up" sectors on disk - where would you put them? Why?
      Should spread them out evenly, so when you replace an arbitrary sector your find one that is close by.

   b. How do you think that the disk controller can check whether a sector has gone bad?
      Using a checksum, this can be efficiently checked in hardware during disk access.

   c. Can you think of any drawbacks of hiding errors like this from the OS?
      Excessive sector failures are warning signs that a disk is beginning to fail.

6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?
   Acknowledge to OS that write has completed after it's written to buffer, not the platters. Write to platter in the background.

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

Break down into seek time, rotation time, transfer time.

Already given seek time of 8 ms.

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

Break down into seek time, rotation time, transfer time.

Already given seek time of 8 ms.

On average, disk must complete ½ revolution to wait for the correct sector to rotate under head. Rotation time is

$$1/2 \times 1 / 7200 \text{ RPM} \cong 4.17 \text{ ms}$$

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

Break down into seek time, rotation time, transfer time.

Already given seek time of 8 ms.

On average, disk must complete ½ revolution to wait for the correct sector to rotate under head. Rotation time is

$$1/2 \times 1 / 7200 \text{ RPM} \cong 4.17 \text{ ms}$$

Transfer time for 4 KiB is

$$4 \text{ KiB} \times 1 / 50 \text{ MiB} \cong 0.078125 \text{ ms}$$

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

Break down into seek time, rotation time, transfer time.

Already given seek time of 8 ms.

On average, disk must complete ½ revolution to wait for the correct sector to rotate under head. Rotation time is

1/2 ✕ 1 / 7200 RPM ≅4.17 ms

Transfer time for 4 KiB is

4 KiB ✕ 1 / 50 MiB ≅ 0.078125 ms

Total time is 8 ms + 4.17 ms + 0.078125 ms ≅ 12.248 ms, giving a throughput of

4 KiB / 12.248 ms ≅ 326.6 KiB/s

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

2. What is the expected throughput of the hard drive when reading 4 KiB sectors from the same track on disk (i.e. the read/write head is already positioned over the correct track when the operation starts)?

3. What is the expected throughput of the hard drive when reading the very next 4 KiB sector (i.e. the read/write head is immediately over the proper track and sector at the start of the operation)?

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

2. What is the expected throughput of the hard drive when reading 4 KiB sectors from the same track on disk (i.e. the read/write head is already positioned over the correct track when the operation starts)?

Ignore seek time, giving a total time of 4.17 ms + 0.078125 ms ≅ 4.24 ms and throughput of

4 KiB / 4.24 ms ≅ 943.4 KiB/s

3. What is the expected throughput of the hard drive when reading the very next 4 KiB sector (i.e. the read/write head is immediately over the proper track and sector at the start of the operation)?

Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

2. What is the expected throughput of the hard drive when reading 4 KiB sectors from the same track on disk (i.e. the read/write head is already positioned over the correct track when the operation starts)?

Ignore seek time, giving a total time of 4.17 ms + 0.078125 ms ≅ 4.24 ms and throughput of

4 KiB / 4.24 ms ≅ 943.4 KiB/s

3. What is the expected throughput of the hard drive when reading the very next 4 KiB sector (i.e. the read/write head is immediately over the proper track and sector at the start of the operation)?

Ignore both rotational and seek times to take full advantage of the transfer rate of 50 MiB/s.