CS162 Midterm 2 Review

Fall 2024

Midterm 2 Logistics

- Tuesday 11/5 from 7-9 PM PST (Election day!)
- Scope:
    - Lectures 1-16, Sections 0-6, HW 0-3, Projects 0-1, and Project 2 Design
- Midterm 2 Logistics
- Midterm 2 Review Session

Disclaimer

This is not an exhaustive review of all topics that are in scope for the midterm.

“You are responsible for the sum total of human knowledge since the beginning of recorded history with particular emphasis on the contents of this course.”

~a certain wise guy at Berkeley

Outline

I. Resource Allocation
   - Deadlock
   - Resource Alloc. Graphs
   - Banker’s Algorithm

II. Scheduling
   - FIFO/RR/SRTF/Priority
   - Realtime

III. Addressing
   - Virtual Memory
   - Paging

IV. Caching
   - Associativity
   - Eviction policies

Resource Allocation

Deadlock & Banker’s Algorithm

Deadlock

What types of resources can a program deadlock on?

All of them (if shared between multiple programs)!
Examples:

- Files
- Memory
- Particular I/O Device
- Locks

We just focus on locks because they're convenient and a common source of deadlock in practice.

- Deadlock ⇒ Starvation but not vice versa

Conditions for Deadlock

1. Hold and wait
   Thread holding at least one resource is waiting to acquire *additional* resources held by other threads.
2. No preemption
   Resources are released only *voluntarily* by the thread holding the resource, *after* thread is finished with it.
3. Mutual exclusion
   Each resource can only be used by one thread at a time.
4. Circular Wait
   There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads s.t.
   - T_1 is waiting for a resource that is held by T_2
   - T_2 is waiting for a resource that is held by T_3
   - …
   - T_n is waiting for a resource that is held by T_1

Conditions for Deadlock

Question:

A: Is it possible to have deadlock without one of these conditions?

B: Does having all four conditions present guarantee deadlock?

Conditions for Deadlock

Question:

A: Is it possible to have deadlock without one of these conditions?

B: Does having all four conditions present guarantee deadlock?

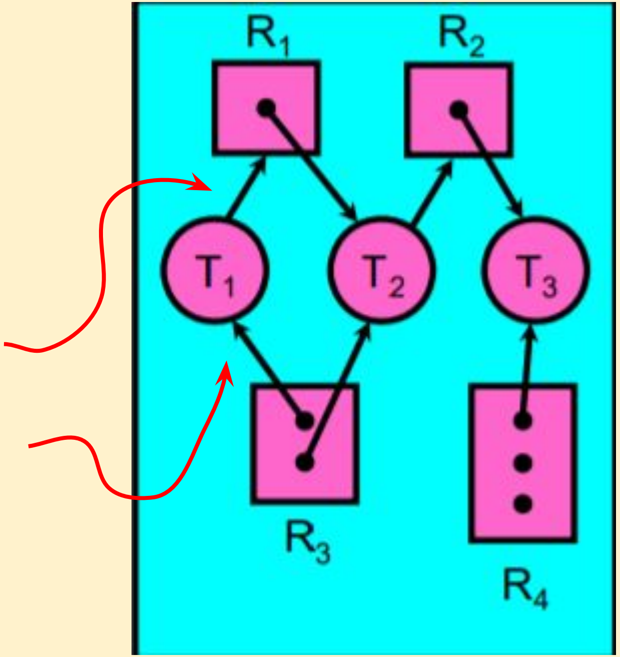
Answer:

A: **No**; all four must be present in order for deadlock to occur.

B: **No**; they are *necessary*, but not *sufficient*.

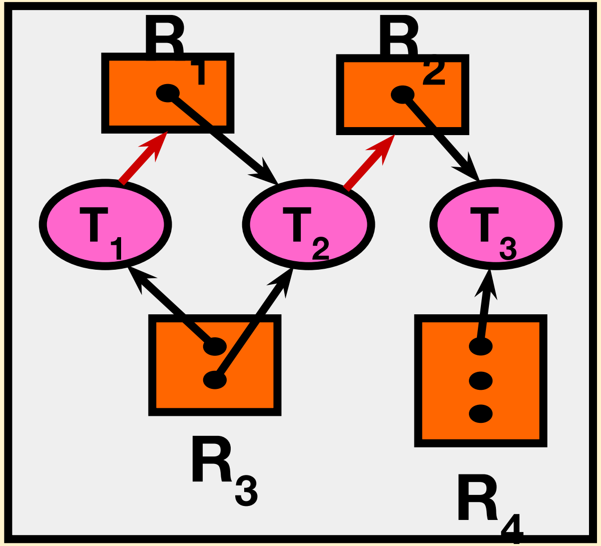
Resource Allocation Graphs

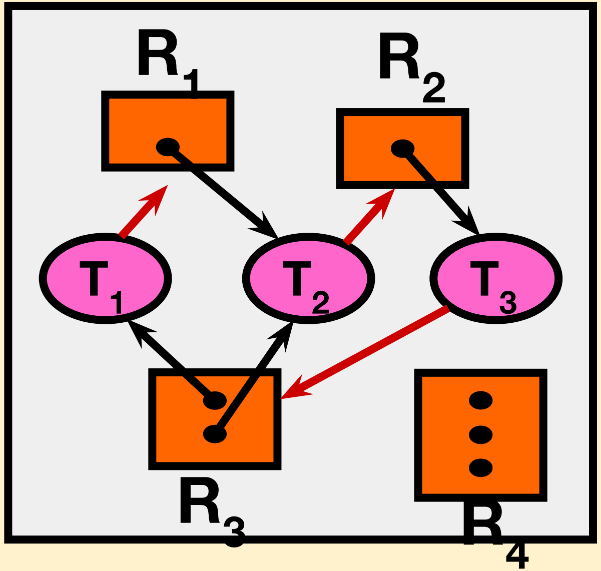
- Model:
    - Set of threads
    - Set of resources
        - Threads compete for access
- Graph structure:
    - Request edge:
      Edge from a thread to a resource it wants to use
    - Assignment edge:
      Edge from a resource to the thread which owns it

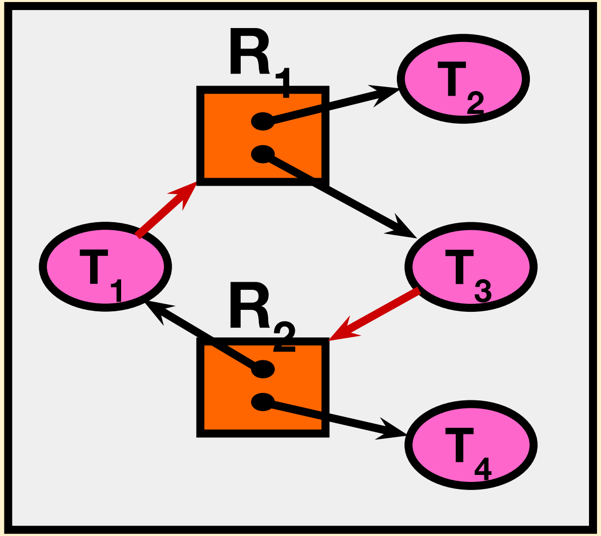


Resource Allocation Graph Examples

- Recall:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

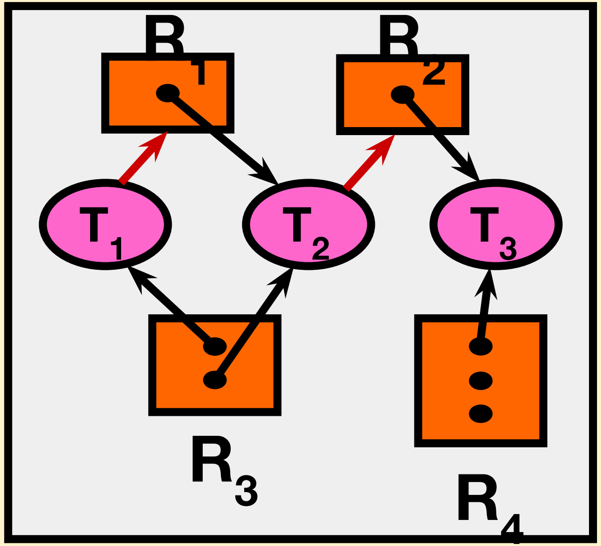




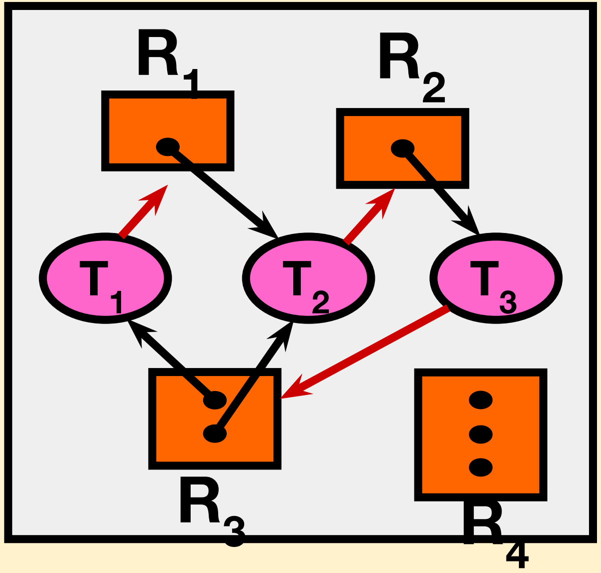


Resource Allocation Graph Examples

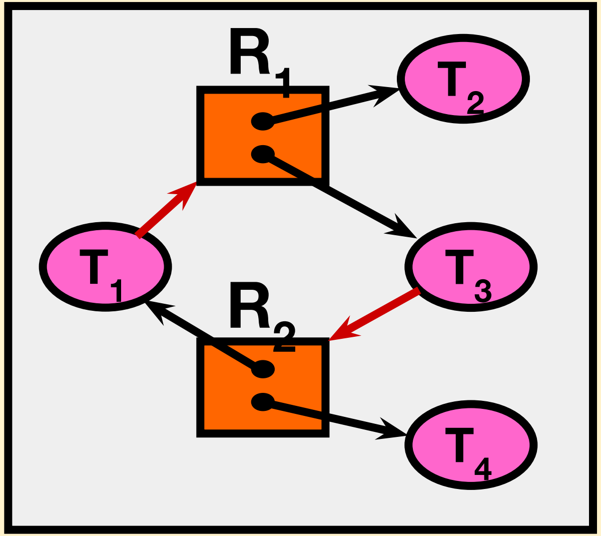
- Recall:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Deadlock: Problem 1

Can this system enter deadlock?

Each function can be called by **any** number of threads.

```
Sema A = init(3);

Sema B = init(1);

Sema C = init(1);

f1() {
    down(A);
    down(B);
    up(A);
    up(B);
}
```

```
f2() {
    down(C);
    down(B);
    up(B);
    up(C);
}
```

```
f3() {
    down(B);
    down(C);
    up(B);
    up(C);
}
```

Deadlock: Problem 1 (Soln)

```
Sema A = init(3);

Sema B = init(1);

Sema C = init(1);
```

Yes. An example of circular wait.

Run: the first line of f1,
first line of f2,
then the first line of f3.

```
f1() {
    down(A);
    down(B);
    up(A);
    up(B);
}
```

```
f2() {
    down(C);
    down(B);
    up(B);
    up(C);
}
```

```
f3() {
    down(B);
    down(C);
    up(B);
    up(C);
}
```

Deadlock: Problem 1 (Soln)

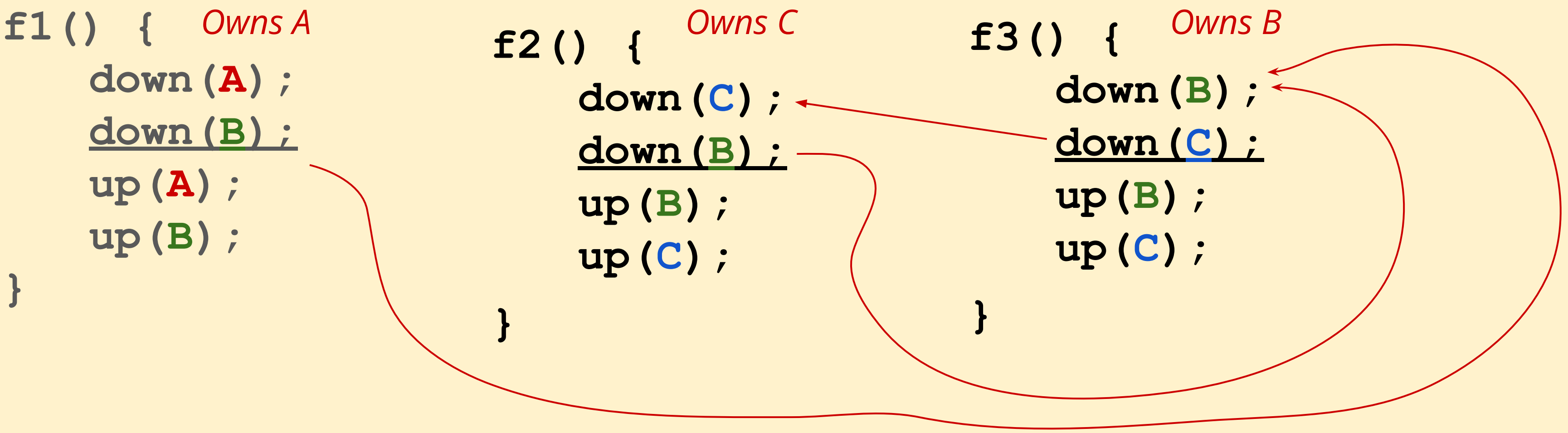
Yes. An example of circular wait.

Run: the first line of f1,
first line of f2,
then the first line of f3.

```
Sema A = init(3);

Sema B = init(1);

Sema C = init(1);
```



Owns A

```
f1() {
    down(A);
    down(B);
    up(A);
    up(B);
}
```

Owns C

```
f2() {
    down(C);
    down(B);
    up(B);
    up(C);
}
```

Owns B

```
f3() {
    down(B);
    down(C);
    up(B);
    up(C);
}
```

Deadlock: Problem 2

```
int matrix[100][100];
Lock xLocks[100];
Lock yLocks[100];
void foo(int x, int y) {
    acquire(xLocks[x]);
    acquire(yLocks[y]);
    matrix[x][y] += 1;
    release(xLocks[x]);
    release(yLocks[y]);
}
```

If multiple threads call this function, can the system enter deadlock? Why or why not?

Deadlock: Problem 2 (soln.)

```
int matrix[100][100];
Lock xLocks[100];
Lock yLocks[100];
void foo(int x, int y) {
    acquire(xLocks[x]);
    acquire(yLocks[y]);
    matrix[x][y] += 1;
    release(xLocks[x]);
    release(yLocks[y]);
}
```

If multiple threads call this function, can the system enter deadlock? Why or why not?

No, because there is no possibility of circular wait:
we always acquire from xLocks before yLocks

Deadlock Avoidance

Some Approaches:

1. Infinite Resources
   a. Virtual Memory
2. Don’t share resources
   a. No IPC
3. Don’t allow waiting
   a. Killing a process
4. Roll back
   a. Journaling
5. Impose ordering
6. Banker's Algorithm

What's the best and potentially easiest to implement?

Deadlock Avoidance

Some Approaches:

1. Infinite Resources
   a. Virtual Memory
2. Don’t share resources
   a. No IPC
3. Don’t allow waiting
   a. Killing a process
4. Roll back
   a. Journaling
5. Impose ordering
6. Banker's Algorithm

What's the best and potentially easiest to implement?

Banker’s Algorithm

The Approach: When a request comes in...

- Pretend the request is granted, *are we at risk of entering deadlock*?
  - If yes, then deny the request (or hang)
  - Else, allow the request

If all threads aren’t able finish, deadlock is possible. This is known as an unsafe state.

Banker’s Algorithm

The Approach: When a request comes in...

- Pretend the request is granted, *are we at risk of entering deadlock*?
    - If yes, then deny the request (or hang)
    - Else, allow the request

If UNFINISHED is not empty, deadlock is possible. This is known as an unsafe state.

```
[Avail] = [Free Resources]
add all threads to UNFINISHED
do {
    DONE = true
    for each NODE in UNFINISHED {
        [Request] = [$Max_{NODE}$] - [$Alloc_{NODE}$]
        if (request <= [Avail]) {
            remove NODE from UNFINISHED
            [Avail] += [$Alloc_{NODE}$]
            DONE = false
        }
    }
} until (DONE)
```

Solving Banker’s Algorithm Problems

Can do whatever works best for you; one possible method:

- Given table of max resource amounts,
  table of resources used per thread:
- Create table of resources needed by each thread to complete,
  table of resources left in the common pool
- See if there are enough resources in the pool to give *any one* of the threads what it needs to complete
  - If no: the program is not in a safe state
  - If yes: ‘run’ the thread, thus returning its resources to the pool;
    - if all threads have completed, the program is in a safe state;
    - otherwise repeat the above until you run out of threads or get blocked

Banker’s Algorithm: Problem

Is this system in a safe state? If this is the case, show a non-blocking sequence of thread executions.

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	1	2	1

Banker's Algorithm: Problem

Solution:

Yes, it is in a safe state.

Ordering: [T3, T1, T2]

*Others exist!

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	1	2	1

Solution:

Yes, it is in a safe state.

Ordering: [T3, T1, T2]

Banker's Algorithm: Problem

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

# Still Needed

	A	B	C
T1	2	0	0
T2	0	1	1
T3	1	1	0

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	1	2	1

	A	B	C
Avail.	1	4	3

Solution:

Yes, it is in a safe state.

Ordering: [T3, T1, T2]

Banker’s Algorithm: Problem

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

# Still Needed

	A	B	C
T1	2	0	0
T2	0	1	1
T3	**1**	**1**	0

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	1	2	1

	A	B	C
Avail.	**1**	**4**	3

Banker’s Algorithm: Problem

Solution:

Yes, it is in a safe state.

Ordering: [T3, T1, T2]

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

# Still Needed

	A	B	C
T1	2	0	0
T2	0	1	1
T3	-	-	-

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	**1 → 0**	**2 → 0**	**1 → 0**

	A	B	C
Avail.	**1 → 2**	**4 → 6**	**3 → 4**

Solution:

Yes, it is in a safe state.

Ordering: [T3, T1, T2]

Banker’s Algorithm: Problem

Max

	A	B	C
T1	3	0	1
T2	0	4	4
T3	2	3	1

# Still Needed

	A	B	C
T1	**2**	0	0
T2	0	1	1
T3	-	-	-

Allocated

	A	B	C
Total	3	9	8
T1	1	0	1
T2	0	3	3
T3	0	0	0

	A	B	C
Avail.	**2**	6	4

Scheduling

Major Scheduling Algorithms

- Shortest Remaining Time First (SRTF)
  - ■ Preemptively schedule process with shortest remaining time to execute
  - + Optimal
  - – Impossible

- First-in, First-out (FIFO)
  - ■ Schedule processes in the order of arrival
  - + Very possible
  - – Unoptimal in certain cases (Convoy effect)

- Strict Priority (Priority)
  - ■ Always run highest priority process
  - + Good for getting important things done
  - – Starvation

- Round-Robin (RR)
  - ■ Run the processes in a looping order for fixed quanta, pre-empting when they’ve used up their time
  - + No starvation
  - – Context switching costs
  - – Have to select quantum

Real-Time Scheduling

- Key points:
    - Ensure system maintains performance guarantees (**Deadlines**)
    - Predictability >> Performance
- Task characteristics
    - Computation times known *in advance* (usually profiled)
    - Tasks have periodic deadlines
- Soft vs. Hard Real Time
    - Soft: *Want to* hit deadlines
        - → Netflix packet streaming
    - Hard: *Must* hit deadlines
        - → Embedded flight control computers

Real-Time Scheduling: Example Algorithms

- Hard Real-time:
  - Earliest Deadline First (*EDF)*
  - Deadline-Monotonic Scheduling (DM)
  - Rate-Monotonic Scheduling (RMS)
  - Least Slack Time Scheduling (LST)

- Soft Real-time
  - Constant Bandwidth Server (CBS)

Other Scheduling Algorithms

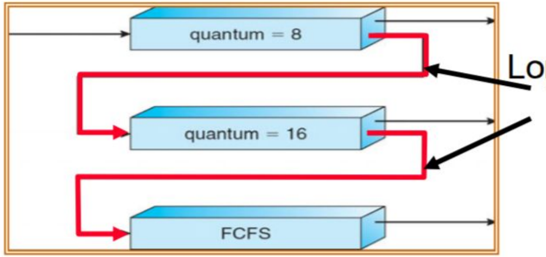
- More Important:
  - Lottery Scheduling
    - ➢ Each process gets a ticket
    - + Avoids starvation
  - Linux Completely Fair Scheduler
    - ➢ Processes have a nice value; higher nice ≡ lower priority
    - ➢ Red black tree to implement ready queue
    - ➢ Pick thread with the smallest vruntime (measure of “CPU runtime”)
    - + CoMpLeTeLy FaIr

- Still Important:
  - Shortest Job First
  - Multilevel Feedback Queue Scheduling

Other Scheduling Algorithms

- More Important:
  - Lottery Scheduling
    - ➢ Each process gets a ticket
    - + Avoids starvation
  - Linux Completely Fair Scheduler
    - ➢ Processes have a nice value; higher nice ≡ lower priority
    - + CoMpLeTeLy FaIr

- Still Important:
  - Shortest Job First
  - Multilevel Feedback Queue Scheduling



Long-Running Compute Tasks Demoted to Low Priority

Abstract/Generalized Scheduling

- (Probably skip this slide tbh)
- Fairness in scheduling
  - Linux CFS
- Scheduling multiple resources
  - Lottery scheduling extension
- Dominant resource fairness (written by Shenker and Stoica!)
    - Sharing incentive
    - strategy-proofness
    - Pareto efficiency
    - Envy-freeness
    - Optimal (extension of max-min fairness)

Every Scheduling Problem Ever (sp17)

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Problem: Schedule & Give Tot. Turnaround Time for each of:

- **Round Robin**
- **SRTF**
- **FIFO**
- **Strict Priority**

- Preemptive priority scheduler
- Break ties in SRTF by priority
- If a process arrives at time x, they are ready to run at the beginning of time x.
- Ignore context switching overhead.
- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list
- Total turnaround time is the time a process takes to complete after it arrives.

Scheduling (RR)

Time:	1	2	3	4	5	6	7	8	9	10	Total Turnaround Time
RR											

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR											

Running

[ ]

Queue

[ ]

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR											

Running

[ ]

Queue

[A]

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**										

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[A]

Queue

[ ]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**										

Running

[ ]

Queue

[A]

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, **add the previously running thread to the ready list**, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 **(B)**	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**										

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, **then add any new threads arriving at quantum X+1** to the ready list

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[A]

[B]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**									

Running

[A]

Queue

[B]

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**									

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[B]

[A]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**								

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[B]

Queue

[A]

[ ]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 **(C)**	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**								

(B is done!)

Running

[ ]

Queue

[A]

[C]

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**							

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[A]

Queue

[C]

[ ]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**							

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[C]

[A]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 **(D)**	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**							

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[C]

[A]

[D]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 **(D)**	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**	**C**						

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[C]

Queue

[A]

[D]

[ ]

Scheduling (RR)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**	**C**	**A**	**D**	**C**	**D**	**D**	

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[ ]

[ ]

[ ]

Scheduling (RR)

A: Arrives @ 1, Finishes @ 6 → Turnaround of **6**
B: Arrives @ 2, Finishes @ 3 → Turnaround of **2**
C: Arrives @ 4, Finishes @ 8 → Turnaround of **5**
D: Arrives @ 5, Finishes @ 10 → Turnaround of **6**

Time:	1 (A)	2 (B)	3	4 (C)	5 **(D)**	6	7	8	9	10	Total Turnaround Time
RR	**A**	**A**	**B**	**A**	**C**	**A**	**D**	**C**	**D**	**D**	**19**

- The quanta for RR is 1 unit of time.
- For round robin: At the end of quantum X, add the previously running thread to the ready list, then add any new threads arriving at quantum X+1 to the ready list

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Running

[ ]

Queue

[ ]
[ ]
[ ]

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF											

Available Processes: {}

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF											

Available Processes: {A: 4}

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	**A**										

Available Processes: {A: 3}

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	A										

Available Processes: {A: 3, B: 1}

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	**A**	**B**									

`Available Processes: {A: 3, B: 0}`

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	**A**	**B**	**A**								

```
Available Processes: {A: 2}
```

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	**A**	**B**	**A**								

```
Available Processes: {A: 2, C: 2}
```

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	A	B	A	C							

Available Processes: {A: 2, C: 1} (break ties)

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	CPU Burst	Arrives at start of	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	A	B	A	C							

`Available Processes: {A: 2, C: 1, D:3}`

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (SRTF)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
SRTF	**A**	**B**	**A**	**C**	**C**	**A**	**A**	**D**	**D**	**D**	**16**

- Preemptive priority scheduler
- Break ties in SRTF by priority

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (FIFO)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
FIFO											

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (FIFO)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
FIFO	**A**	**A**	**A**	**A**	**B**	**C**	**C**	**D**	**D**	**D**	**18**

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Scheduling (Priority)

Time:	1 (A)	2 (B)	3	4 (C)	5 (D)	6	7	8	9	10	Total Turnaround Time
Priority	**A**	**B**	**A**	**C**	**C**	**D**	**D**	**D**	**A**	**A**	**17**

- Preemptive priority scheduler

Process	**CPU Burst**	**Arrives at start of**	**Priority**
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Take a break :)

Addressing

address translation & virtual memory

Units to Remember!

- **Numbers**:
  - 1 B = 8 bits
  - 1 KB = 2^10 B (not 1000!!!!)
  - 1 MB = 2^20 B
  - 1 GB = 2^30 B
  - 1 TB = 2^40 B
- **Units of Time**:
  - 1 ms = 10^-3 s
  - 1 µs = 10^-6 s
  - 1 ns = 10^-9 s
  - 1 ps = 10^-12 s

Why do we need Virtual Memory?

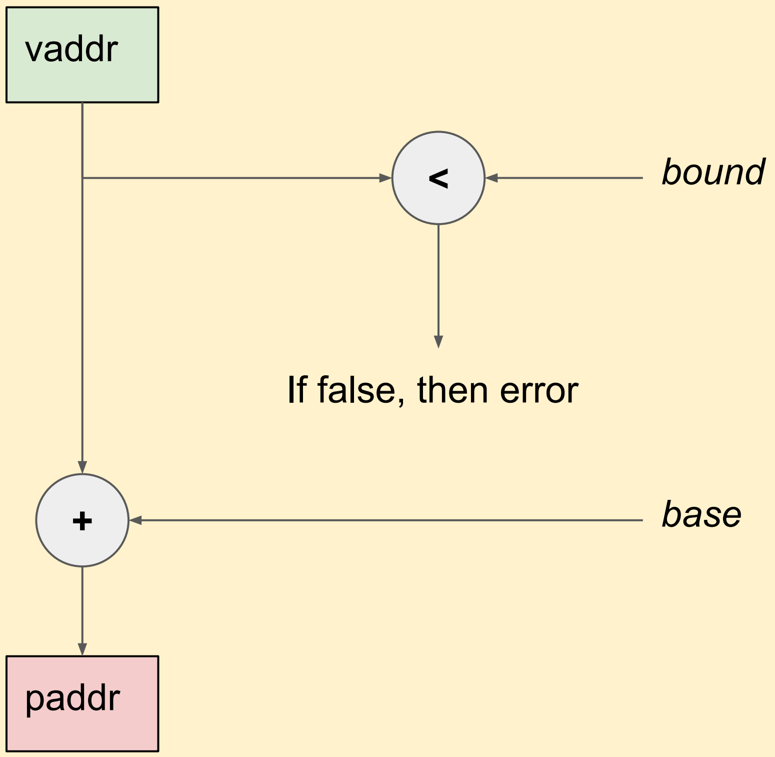
1. Protection: A process cannot access another process’s memory
2. Translation: Processor uses virtual addresses, Physical memory uses physical addresses
3. Efficiency: allows us to avoid paging out all of a process’ memory when taking it off of the CPU

Addressing Schemes

In increasing order of complexity:

- Base & Bound
- Memory Segmentation
- Paging
    - Single-Level
    - Multi-Level
    - Inverted (Linear)
    - Inverted (Hashed)

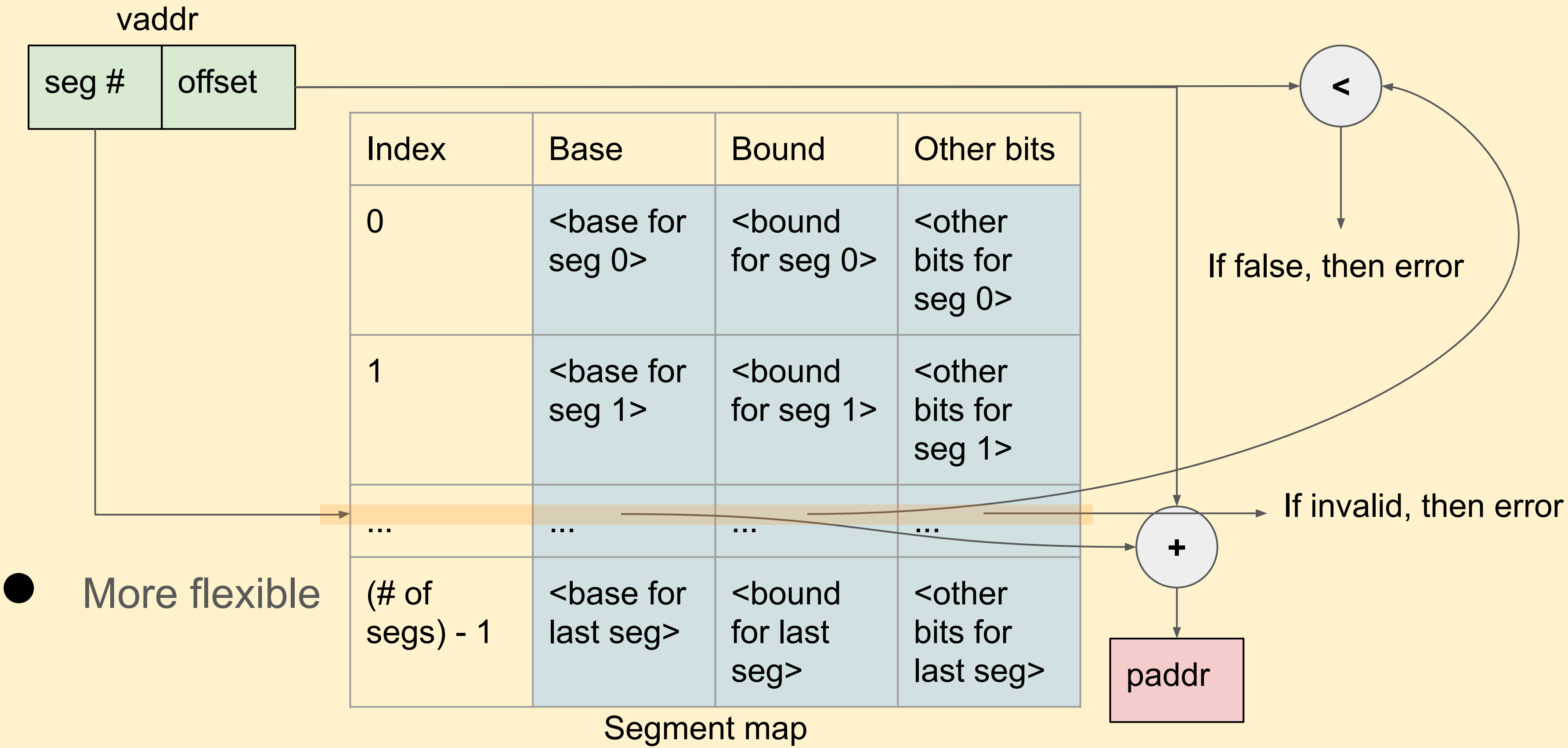
Base and Bound



- Pros:
  - + Simple and fast protection & isolation
  - + Can easily relocate program

- Cons:
  - – Promotes internal & external fragmentation
  - – Inter-process sharing is difficult

(x86) Memory Segmentation



Index	Base	Bound	Other bits
0	<base for seg 0>	<bound for seg 0>	<other bits for seg 0>
1	<base for seg 1>	<bound for seg 1>	<other bits for seg 1>
...	...	...	...
(# of segs) - 1	<base for last seg>	<bound for last seg>	<other bits for last seg>

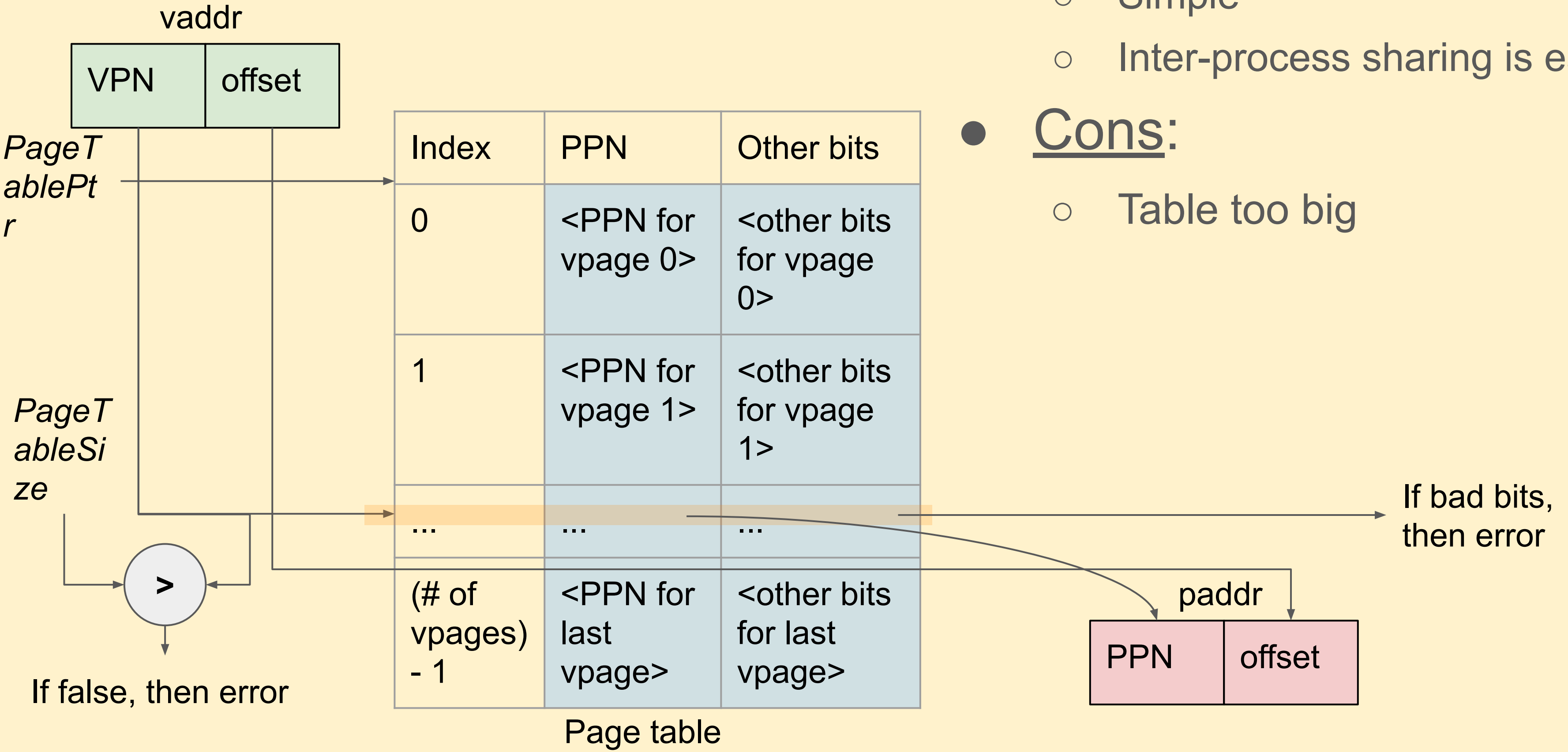
Segment map

- More flexible

Problems? Can we do better?

- Some fragmentation with variable-sized chunks in physical memory
- Move around processes a lot
- Translation happens on every single instruction
- Solution: Pages!
  - A new unit of memory where the physical address space is divided into fixed-sized chunks
  - Now physical memory ⇔ an array of pages
  - Typically pages around ~1k-16k to avoid internal fragmentation

Page Table: Single-Level



Index	PPN	Other bits
0	<PPN for vpage 0>	<other bits for vpage 0>
1	<PPN for vpage 1>	<other bits for vpage 1>
...	...	...
(# of vpages) - 1	<PPN for last vpage>	<other bits for last vpage>

Page table

- Pros:
    - Simple
    - Inter-process sharing is easy
- Cons:
    - Table too big

Some terminology

- Virtual Page Number (VPN): Maps 1-to-1 with an entry in the page table
- Physical Page Number (PPN): Location in physical memory of the page
    - PTE = (# PPN bits) + (# control bits)
- Offset: Points to specific bytes in the physical page
    - 2^(# offset bits) ⇔ size of pages!

Paging Understanding Check!

1) What is contained in each entry of the page table?
2) How do we figure out how many entries in our page table there are?
3) T/F, paging produces less external fragmentation than base and bound.
4) What happens when I access memory marked as invalid?
5) How much memory does a page table with a 22-bit VPN with a 10-bit offset take?

Paging Understanding Check!

1) What is contained in each entry of the page table?
   1. The PPN
   2. Control bits (R, W, X, Valid, Dirty, Use, …)
      (also called access bits, status bits, metadata bits, ‘other bits’)
2) How do we figure out how many entries in our page table there are?
   - Number of possible VPNs (i.e. 2^(VPN #bits))
3) T/F, paging produces less external fragmentation than base and bound.
   T
4) What happens when I access memory marked as invalid?

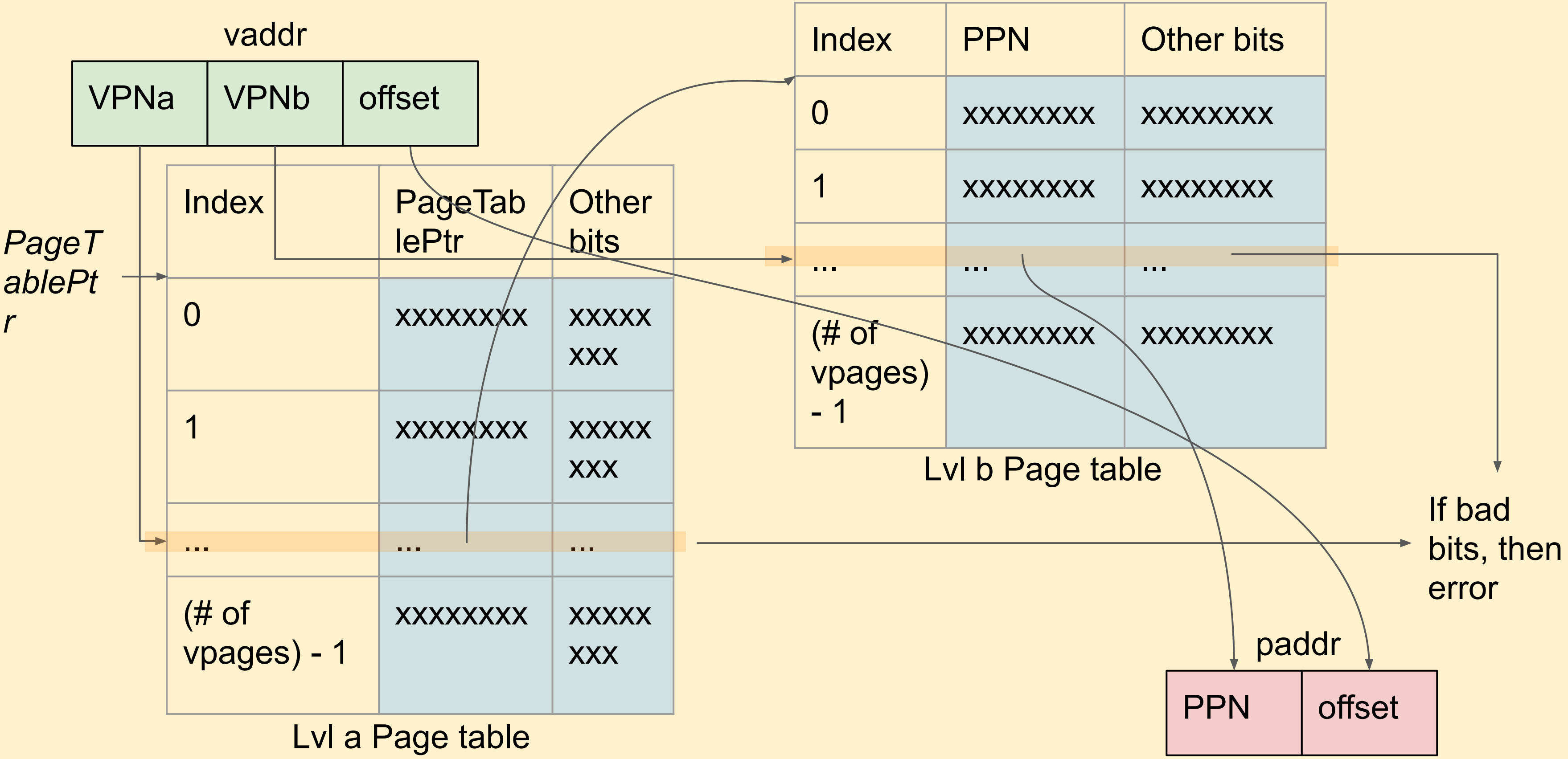
   The page fault handler is run.
5) How much memory does a page table with a 22-bit VPN with a 10-bit offset take?

   22-bits is 4 million entries => 16 MB!

Page Table: Multi-Level

Pro:

- Smaller tables



vaddr

VPNa	VPNb	offset

PageTablePtr

Index	PageTablePtr	Other bits
0	xxxxxxxx	xxxxxxxx
1	xxxxxxxx	xxxxxxxx
...	...	...
(# of vpages) - 1	xxxxxxxx	xxxxxxxx

Lvl a Page table

Index	PPN	Other bits
0	xxxxxxxx	xxxxxxxx
1	xxxxxxxx	xxxxxxxx
...	...	...
(# of vpages) - 1	xxxxxxxx	xxxxxxxx

Lvl b Page table

If bad bits, then error

paddr

PPN	offset

Multi-level Understanding Check!

Let’s say I still have a 32 bit virtual address space and 4 KiB pages, but I break the VPN up into 10 bits and 10 bits for use in a two level page table. How big are my page tables put together if I only have one page mapped?

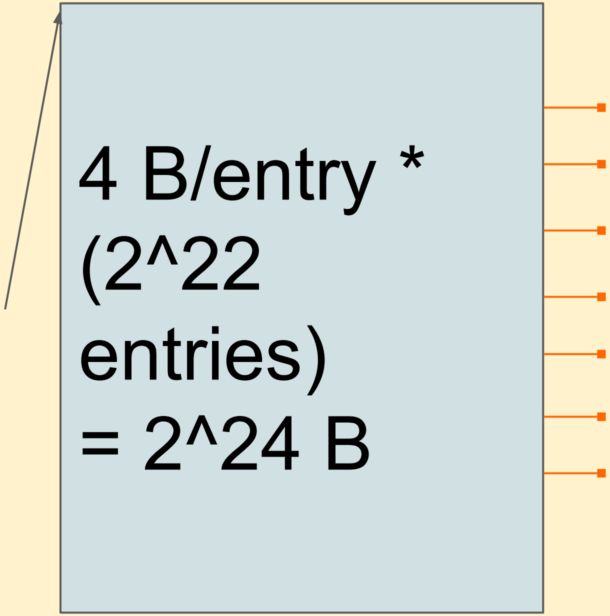
Multi-level Understanding Check!

Let’s say I still have a 32 bit virtual address space and 4 KiB pages, but I break the PPN up into 10 bits and 10 bits for use in a two level page table. How big are my page tables put together if I only have one page mapped?

- If you only have one page mapped then you only need one node in the first layer and one node in the second layer. 10 bits for each level means 2^10 entries in each node. Because each entry is 4 bytes, 2^10 entries fits perfectly in one page (each node is one page). There are two nodes, so two pages = 4KiB + 4KiB = 8KiB memory needed. Huge improvement from 4MiB!

Multi-level Understanding Check!

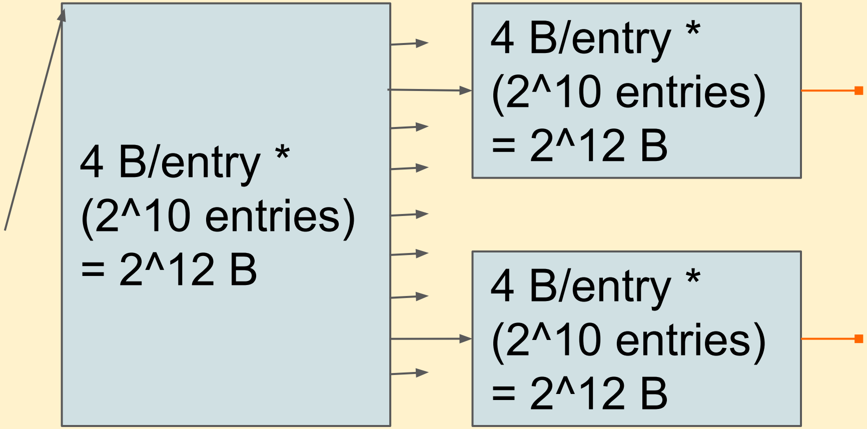
Single-level page table



4 B/entry * (2^22 entries) = 2^24 B

Total: 16MiB

Multi-level page tables



4 B/entry * (2^10 entries) = 2^12 B

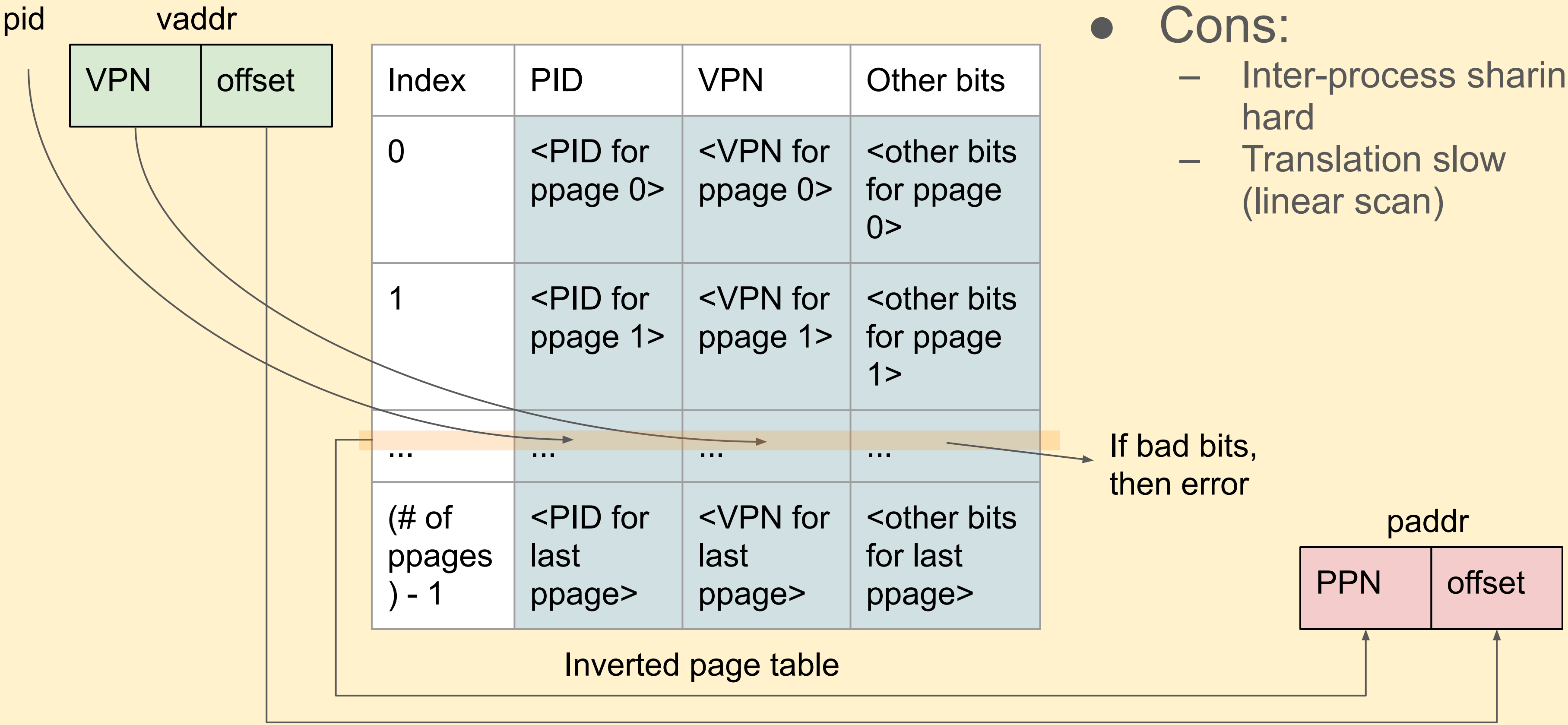
4 B/entry * (2^10 entries) = 2^12 B

4 B/entry * (2^10 entries) = 2^12 B

Total: 12KiB

(Linear) Inverted Page Table

- Pros:
  - + Single table for all processes → small
- Cons:
  - – Inter-process sharing hard
  - – Translation slow (linear scan)



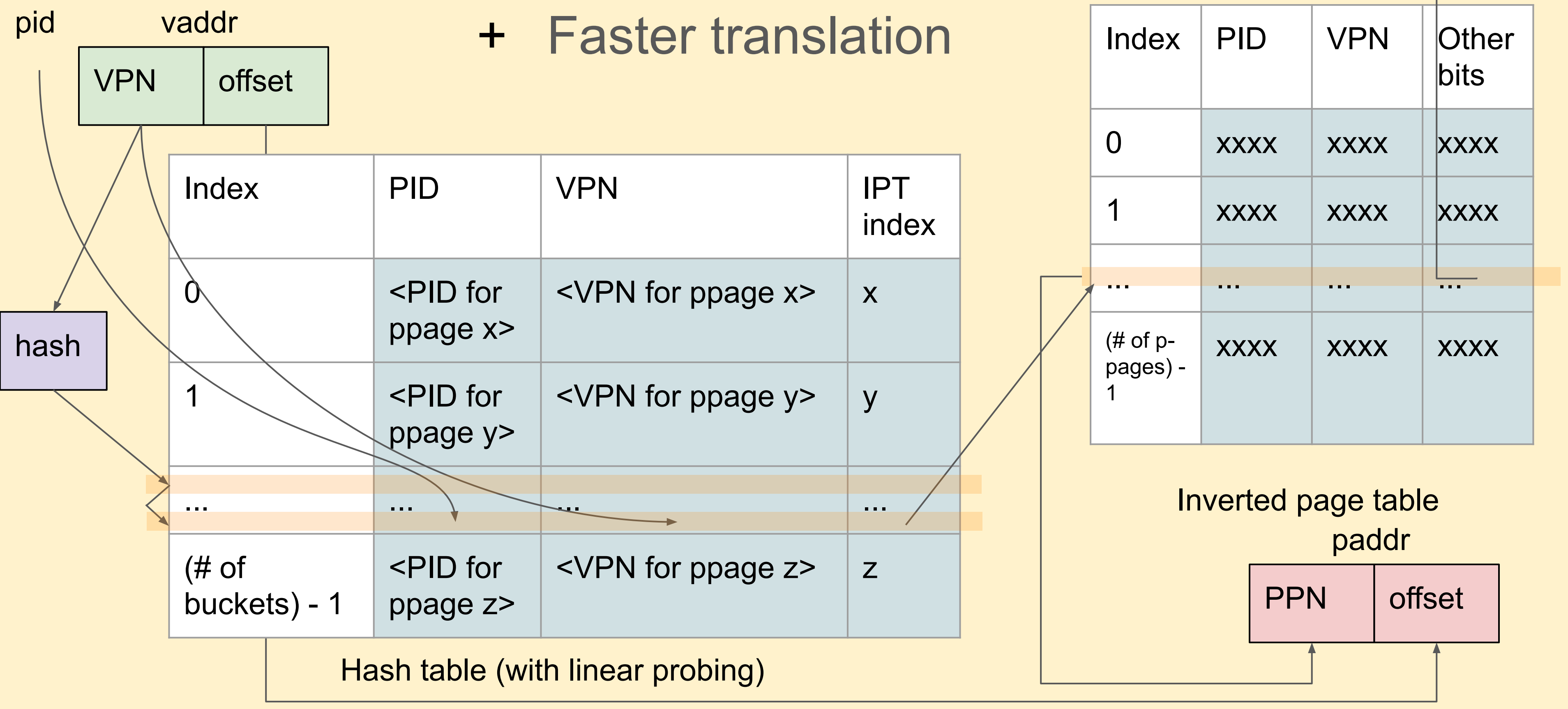
Index	PID	VPN	Other bits
0	<PID for ppage 0>	<VPN for ppage 0>	<other bits for ppage 0>
1	<PID for ppage 1>	<VPN for ppage 1>	<other bits for ppage 1>
...	...	...	...
(# of ppages) - 1	<PID for last ppage>	<VPN for last ppage>	<other bits for last ppage>

Inverted page table

(Hashed) Inverted Page Table

+ Faster translation

If bad bits, then error



Index	PID	VPN	IPT index
0	<PID for ppage x>	<VPN for ppage x>	x
1	<PID for ppage y>	<VPN for ppage y>	y
...	...	...	...
(# of buckets) - 1	<PID for ppage z>	<VPN for ppage z>	z

Hash table (with linear probing)

Index	PID	VPN	Other bits
0	xxxx	xxxx	xxxx
1	xxxx	xxxx	xxxx
...	...	...	...
(# of p-pages) - 1	xxxx	xxxx	xxxx

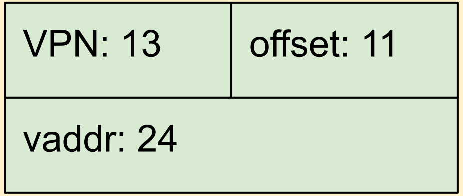
Inverted page table

2018 Fall MT2 P5.a Derivative

- 24 bit virtual address space
- 2 KiB page size
- Single-level page table
- 2 B page table entries
- Q: how many bits is VPN; how many bits is offset?

2018 Fall MT2 P5.a Derivative

- Q: how many bits is VPN; how many bits is offset?
- A: 13; 11 because
  - offset #bits = lg(2 Ki) = 11
  - VPN #bits = (virtual addr space #bits) - (offset #bits) = 24 - 11 = 13



VPN: 13

offset: 11

vaddr: 24

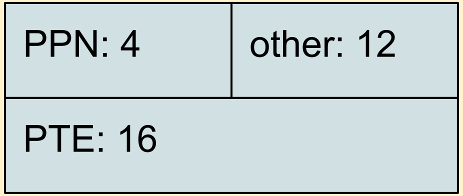
2018 Fall MT2 P5.b Derivative

- 24-bit virtual address space
- 2 KiB page size
- Single-level page table
- 2 B page table entries
- 13 bit VPN; 11 bit offset

Problem:
Given that the PTE will contain 12 control bits per entry;
What is *max* possible size of the physical address space?

2018 Fall MT2 P5.b Derivative

- Given that the PTE will contain 12 control bits per entry; What is *max* possible size of the physical address space?
- A: **15-bit** because
  - PPN #bits = (PTE #bits) - (# other bits) = 2 * 8 - 12 = 16 - 12 = 4
  - physical address space #bits = (PPN #bits) + (offset #bits) = 4 + 11 = 15



PPN: 4

other: 12

PTE: 16

2018 Fall MT2 P5 Derivative

- 24-bit virtual address space
- 2 KiB page size
- Single-level page table
- 2 B page table entries
- 13 bit VPN; 11 bit offset
- **Max 4 bit PPN; 12 other PTE bits (control bits)**

2018 Fall MT2 P5.c Derivative

- **13** bit VPN; **11** bit offset
- Max **4** bit PPN
- **12** other/control bits
- Assume this is **big-endian**

PPN: 4	other: 12
PTE: 16	

Problem:

Assuming we have 8 KiB physical memory w/ the PTE layout and memory on the right, and PageTablePtr as 0x8, translate 0x000844, 0x000488, 0x000ccc

*Assume valid bits & such check out fine

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- **13** bit VPN; **11** bit offset
- Max **4** bit PPN
- **12** other/control bits
- Assume this is **big-endian**

PPN: 4	other: 12
PTE: 16	

Problem:

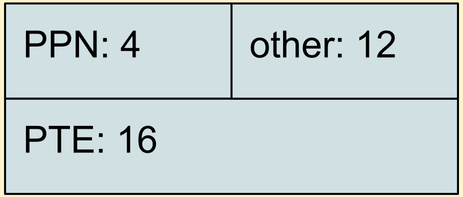
Assuming we have 8 KiB physical memory w/ the PTE layout and memory on the right, and PageTablePtr as 0x8, translate 0x000844, 0x000488, 0x000ccc

Soln: 0x5844, 0x6c88, 0xbccc

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- **13** bit VPN; **11** bit offset
- Q: *PageTablePtr* as 0x8,
  translate **0x000844**


PPN: 4

other: 12

PTE: 16

Answer: **0x5844** because

- vaddr: 0b1000 0100 0100
- VPN: 1; offset: 0b000 0100 0100
- PTE @ (*PageTablePtr* + VPN * (PTE size in bytes)) = 0x8 + 1 * 2 = 0xa
- PTE: 0b1011 0000 1111 1010
- PPN: 0b1011 = b
- paddr: PPN || offset = 0b1011 || 0b000 0100 0100 = 0x5844

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- **13** bit VPN; **11** bit offset
- Q: *PageTablePtr* as 0x8, translate **0x000488**

Answer: **0x6c88** because

- VPN: 0; offset: 0b100 1000 1000
- PTE @ (0x8 + 0 * 2) = 0x8
- PTE: 0b1101 1111 1100 0101
- PPN: 0b1101 = d
- paddr: 0b1101 || 0b100 1000 1000 = 0x6c88

PPN: 4	other: 12
PTE: 16	

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- **13** bit VPN; **11** bit offset
- Q: *PageTablePtr* as 0x8, translate **0x000ccc**

PPN: 4	other: 12
PTE: 16	

Answer: **0x5ccc** because

- VPN: 1; offset: 0b100 1100 1100
- PTE @ (0x8 + 1 * 2) = 0xa
- PTE: 0b1011 0000 1111 1010
- PPN: 0b1011 = b
- paddr: 0b1011 || 0b100 1100 1100 = 0x5ccc

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- 13 bit VPN; 11 bit offset
- Max 4 bit PPN; 12 other PTE bits (control bits)
- Q: assuming we have 16 KiB physical memory, PTE layout and memory on the right, and *PageTablePtr* as 0x4, translate 0x0008ad, 0x002655, 0x001ff1 (also assume bits aside from valid always check out good)

unused: 1	PPN: 3	other: 11	valid: 1
PTE: 16			

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- 13 bit VPN; 11 bit offset
- Max 4 bit PPN; 12 other PTE bits (control bits)
- Q: assuming we have 16 KiB physical memory, PTE layout and memory on the right, and *PageTablePtr* as 0x4, translate 0x0008ad, 0x002655, 0x001ff1 (also assume bits aside from valid always check out good)
- A: 0x10ad, 0x1e55, page fault

unused: 1	PPN: 3	other: 11	valid: 1
PTE: 16			

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.c Derivative

- 13 bit VPN; 11 bit offset
- Q: *PageTablePtr* as 0x4, translate 0x001ff1
- A: page fault because
  - vaddr: 0b1 1111 1111 0001
  - VPN: 3; offset: 0b111 1111 0001
  - PTE @ (0x4 + 3 * 2) = 0xa
  - PTE: 0b1011 0000 1111 1010
  - valid: 0

unused: 1	PPN: 3	other: 11	valid: 1
PTE: 16			

Address	+0	+1	+2	+3
0x00	fe	91	f6	a9
0x04	93	8c	a8	d7
0x08	df	c5	b0	fa
0x0c	bc	83	c4	dd
0x10	9d	af	88	ae
0x14	b8	e9	9d	f0

2018 Fall MT2 P5.d Derivative

- 24-bit virtual address space
- 2 KiB page size
- 2 B page table entries
- 11 bit offset
- Max 4 bit PPN; 12 other PTE bits (control bits)

Problem:

Now suppose we want to transform our single level page table into a multi-leveled page table.

Assuming that every page table is required to fit into a single page, how many total levels of page tables do we need to address the entire virtual address space?

2018 Fall MT2 P5.d Derivative

Question: Now suppose we want to transform our single level page table into a multi-leveled page table.
Assuming that every page table is required to fit into a single page, how many total levels of page tables do we need to address the entire virtual address space?

Answer: **2**

- **#PTE per page** = (page size) / (PTE size) = (2^11 B) / (2 B) = 2^10
- **Max level x VPN #bits** = lg(#PTE per page) = lg(2^10) = 10
- **vaddr bits** = 24.
- We need (total # VPN bits) + (# offset bits) >= 24 to address the entire virtual address space. With a max of 10 VPN bits per page and x pages:
  - 10 * x (# VPN bits) + 11 (# offset bits) >= 24
  - The smallest x that satisfies this is x = 2. Thus, we need at least **2 levels**.

2016 Fall MT2 3.c Derivative

Consider a hashed inverted page table and two processes P1 and P2. We use an 8-bit virtual address space and 4 byte page size. The hash table uses a hash function which simply calculates the index by adding PID and VPN mod 8. The IPT has its next free entry at index 4.

i) **Translate the accesses: (P1, 0x3), (P1, 0x28), (P1, 0xee).**

ii) What happens upon read access (P2, 0x84)?

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6			
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4			
...			

Inverted page table

2016 Fall MT2 3.c Derivative (Soln.)

Consider a hashed inverted page table and two processes P1 and P2. We use an 8-bit virtual address space and 4 byte page size. The hash table uses a hash function which simply calculates the index by adding PID and VPN mod 8. The IPT has its next free entry at index 4.

i) Translate the accesses: (P1, 0x3), (P1, 0x28), (P1, 0xee). A: 0xF, 0x0, 0xA

ii) What happens upon read access (P2, 0x84)? A: page fault, update tables

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6	2	0x21	4
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4	2	0x21	VR
...			

Inverted page table

2016 Fall MT2 3.c Derivative

i) Translate the accesses:

(P1, 0x3), (P1, 0x28), (P1, 0xee).

A: 0xF, **0x0**, 0xA because

1. `0x3` = `0b0011` ⇒ (VPN = `0b00`) & (Offset = `0b11`) ⇒ (VPN = `0x0`) & (Offset = `0x3`) ⇒ (Hash Value = 1)

2. (PID: 1, VPN: `0x0`) at hash table idx 1 matches, so IPT idx is 3
   ⇒ `0b11` PPN
   ⇒ paddr = `0b11` || `0b11` = `0xf`

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6			
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4			
...			

Inverted page table

2016 Fall MT2 3.c Derivative

i) Translate the accesses:

(P1, 0x3), **(P1, 0x28)**, (P1, 0xee).

A: 0xF, **0x0**, 0xA because

1. `0x28` = `0b101000` ⇒ (VPN = `0b1010`)& `0b00` offset ⇒ (VPN = `0xA`) & (Offset = `0x0`) ⇒ hash value =
   (`0b1010` + 1) mod 8 =
   `0b1011` mod 8 = `0b011` = 3
2. HT entry 3 matches PID and VPN, so IPT idx is 0
   ⇒ PPN = `0b0`
   ⇒ paddr = (`0b0` || `0b00`) = `0x0`

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6			
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4			
...			

Inverted page table

2016 Fall MT2 3.c Derivative

i) Translate the accesses:

(P1, 0x3), (P1, 0x28), **(P1, 0xee)**.

A: 0xF, 0x0, **0xA** because

1. 0xEE = 0b11101110 ⇒
   (VPN = 0b111011) & (Offset = 0b10) ⇒
   (VPN = 0x3B) & (Offset = 0x2) ⇒
   Hash Value = (0b111011 + 1) mod 8 =
   0b111100 mod 8 = 0b100 = 4
2. HT #4 does not match PID and VPN!
3. HT entry 5 matches PID and VPN, so
   IPT idx is 2 ⇒ 0b10 PPN ⇒
   paddr = (0b10 || 0b10) = 0xA

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6			
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4			
...			

Inverted page table

2016 Fall MT2 3.c Derivative

ii) What happens upon read access (P2, 0x84)?

A: page fault, update tables because

1. `0x84` = `0b10000100` ⇒ VPN = `0b100001`
   `0b00` offset ⇒ `0x21` VPN
   `0x0` offset ⇒ hashval =
   (`0b100001` + 2) mod 8 =
   `0b100011` mod 8 = 3
2. HT entry 3 does not match
3. HT entry 4, 5 also fail
4. HT entry 6 is empty ergo page fault
   - Allocate ppage 4, and update tables accordingly

idx	PID	VPN	IPT idx
0			
1	1	0x0	3
2			
3	1	0xA	0
4	2	0x3A	1
5	1	0x3B	2
6	2	0x21	4
7			

Hash table (with linear probing)

idx	PID	VPN	other bits
0	1	0xA	VRW
1	2	0x3A	VR
2	1	0x3B	VRW
3	1	0x0	VRW
4	2	0x21	4
...			

Inverted page table

2023 Spring MT2 P5.c

Question: In class, we discussed the “magic” address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

[VPN1: 10-bits | VPN2: 10-bits | Offset: 12-bits]

- **What is “magic” about this configuration?**

2023 Spring MT2 P5.c

Question: In class, we discussed the “magic” address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

[VPN1: 10-bits | VPN2: 10-bits | Offset: 12-bits]

- **What is magic about this configuration?**

Answer:

- Each level of the 2-level page table takes up exactly 1 page in size. 2^(# offset bits) = 2^12 = 4 KB pages.
    - 10-bit = 1024 entries. 4 B entries => 4 KB per page table!

2023 Spring MT2 P5.d

Question: Assume that we have a 64-bit processor which has the same page size as you gave in problem (5a) and the same 12 access control bits as given in the above PTE.

1) Now, if we reserve 8-bytes for each PTE, how would the virtual address be divided for a 64-bit address space to preserve magic (except maybe highest level)?
2) How many levels of page table would this imply?
3) Bonus: why do we choose the highest level to not necessarily map to a full page?

2023 Spring MT2 P5.d

Question: Assume that we have a 64-bit processor which has the same page size as you gave in problem (5a) and the same 12 access control bits as given in the above PTE.

1) Now, if we reserve 8-bytes for each PTE, how would the virtual address be divided for a 64-bit address space to preserve magic (except highest level)?
2) How many levels of page table would this imply?
3) Bonus: why do we only choose the highest level to not necessarily map to a full page?

Answer:

1) Instead of 10-bit VPN, a 9-bit VPN => 4 KB page (since PTE = 8 bytes). With 12 offset bits, 52 bits for PPN => 7/9/9/9/9/9/12.
2) This is a 6-level page table!
3) This ensures that the magic property holds for as many page tables as possible!

Caching

Cache Basics

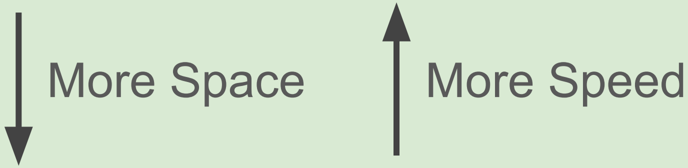
- Caches contain copies of data that can be accessed faster than from memory
- Locality
    - Temporal Locality:

      Recently accessed data close to processor
    - Spatial Locality:

      Contiguous blocks are close to each other
- AMAT: Hit Rate x Tc + Miss Rate x (Tc + Miss Penalty)
    - Tc = cache access time

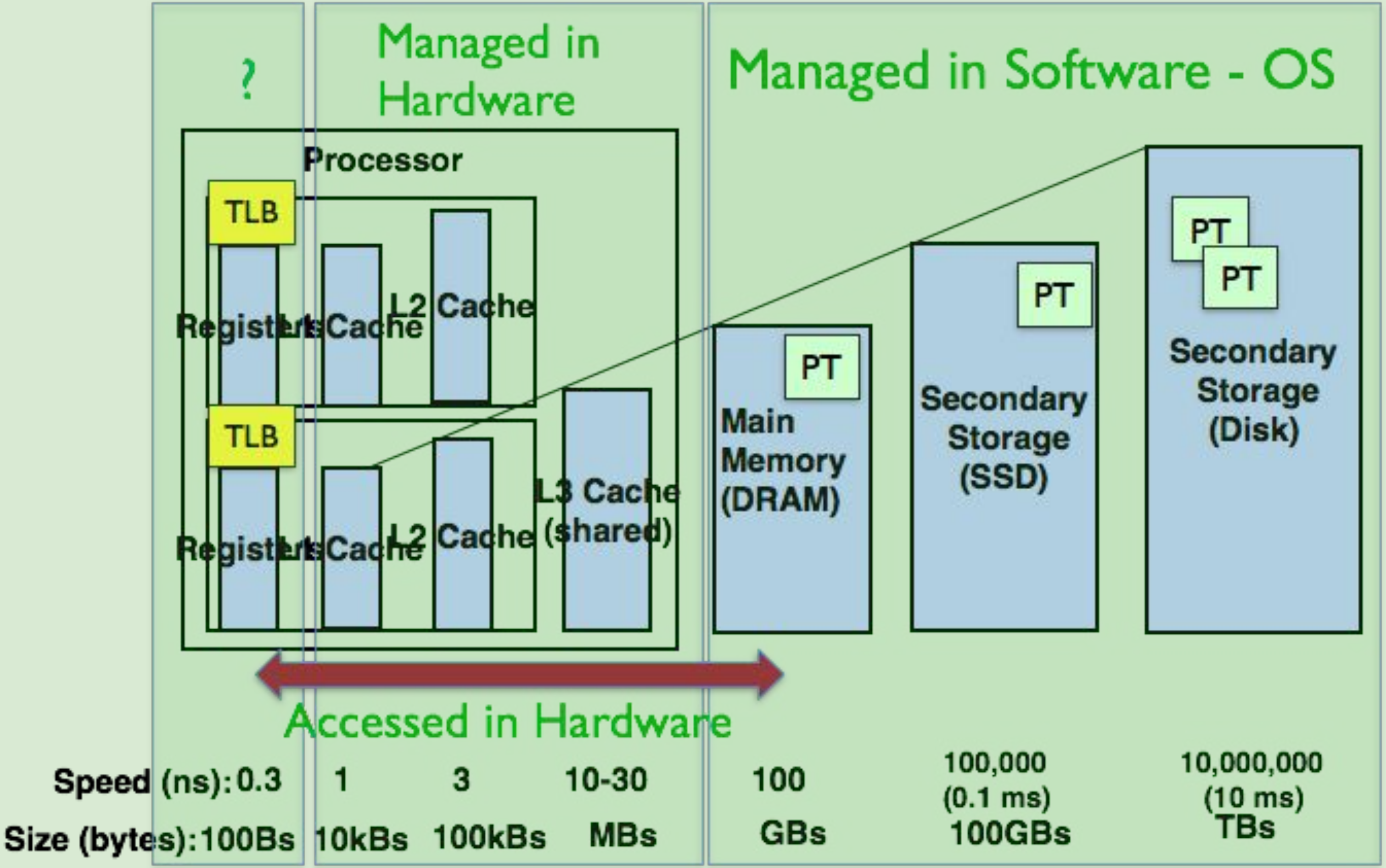
Memory Hierarchy

- Registers
- L1 Cache
- L2, L3... Caches
- Main Memory
- Secondary Storage



More Space

More Speed



?

Managed in Hardware

Managed in Software - OS

Processor

TLB

Registers

L1 Cache

L2 Cache

TLB

Registers

L1 Cache

L2 Cache

L3 Cache (shared)

PT

Main Memory (DRAM)

PT

Secondary Storage (SSD)

PT

PT

Secondary Storage (Disk)

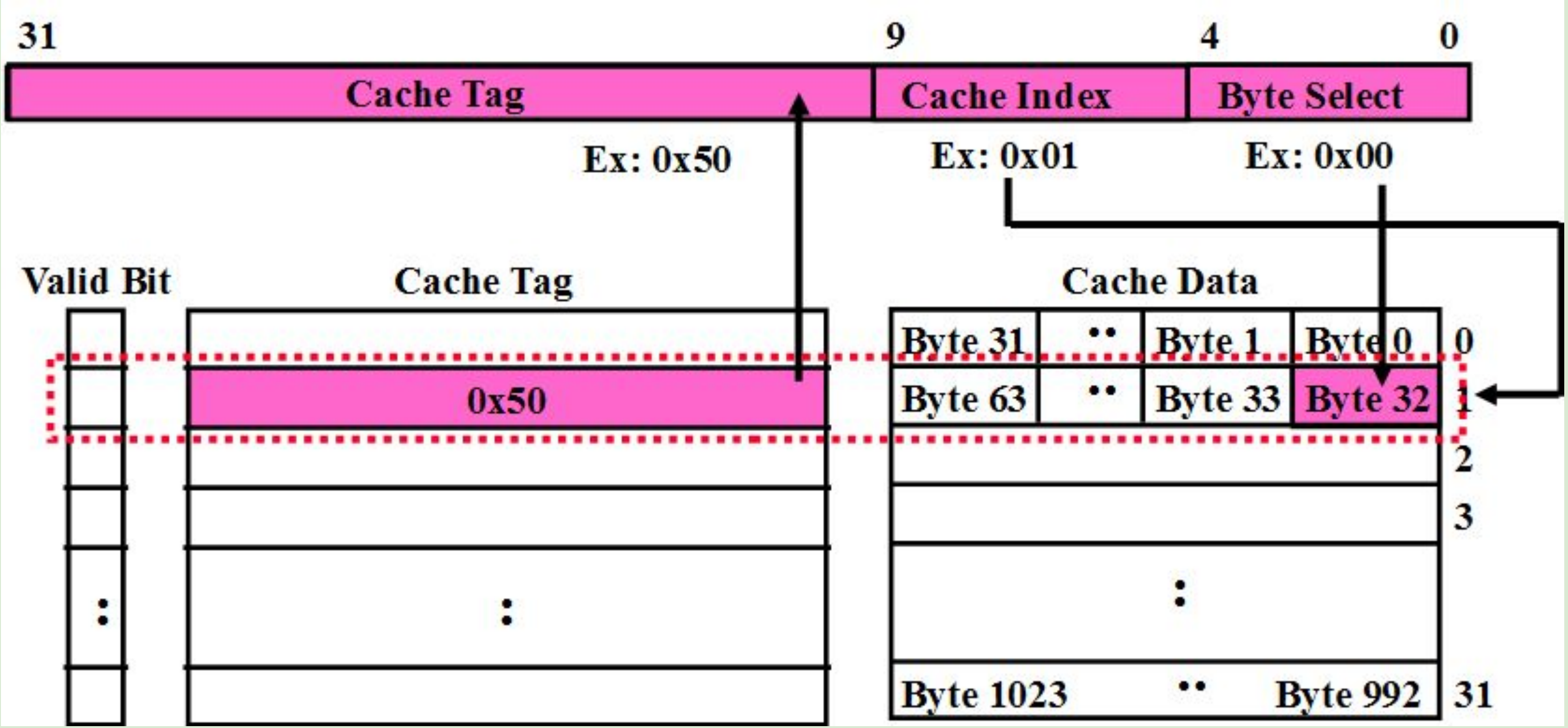
Accessed in Hardware

Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

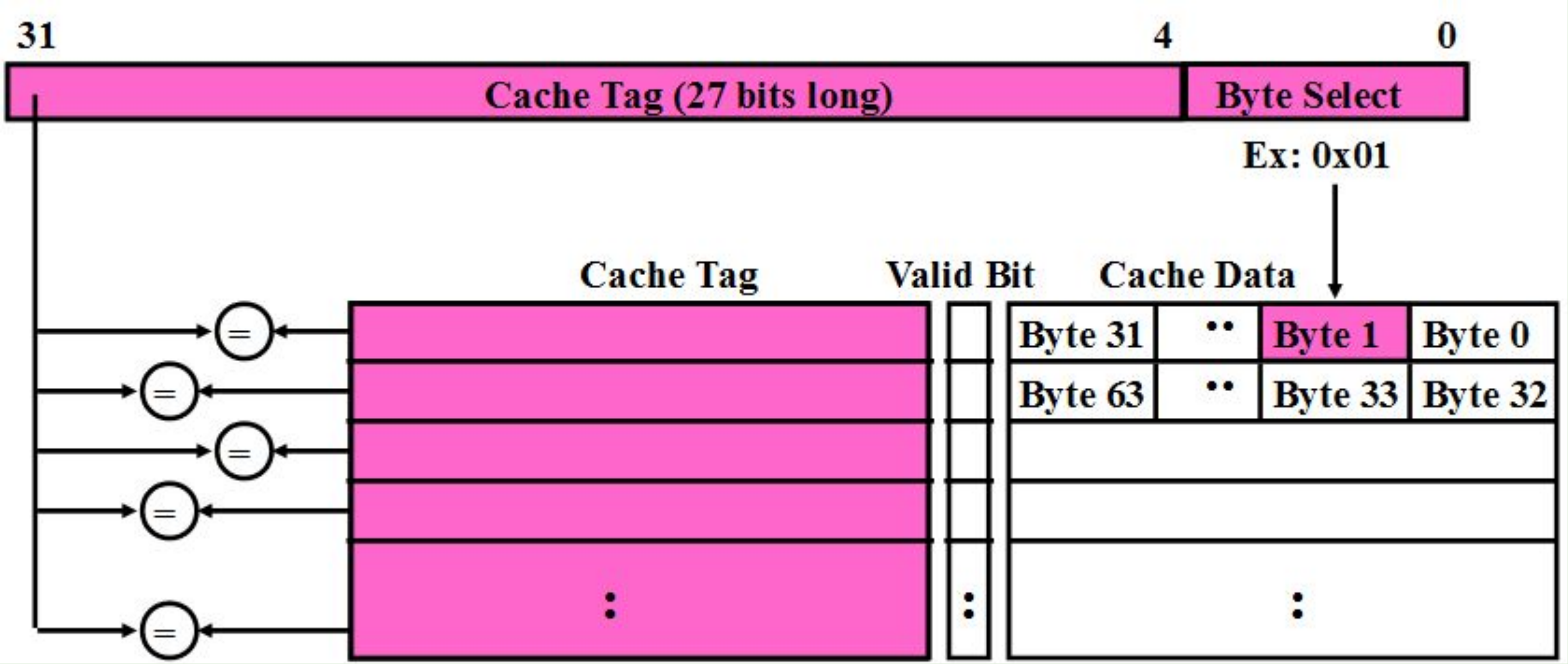
Caches

- Types of Caches:
  - Direct Mapped
  - Fully Associative
  - N-way Set Associative

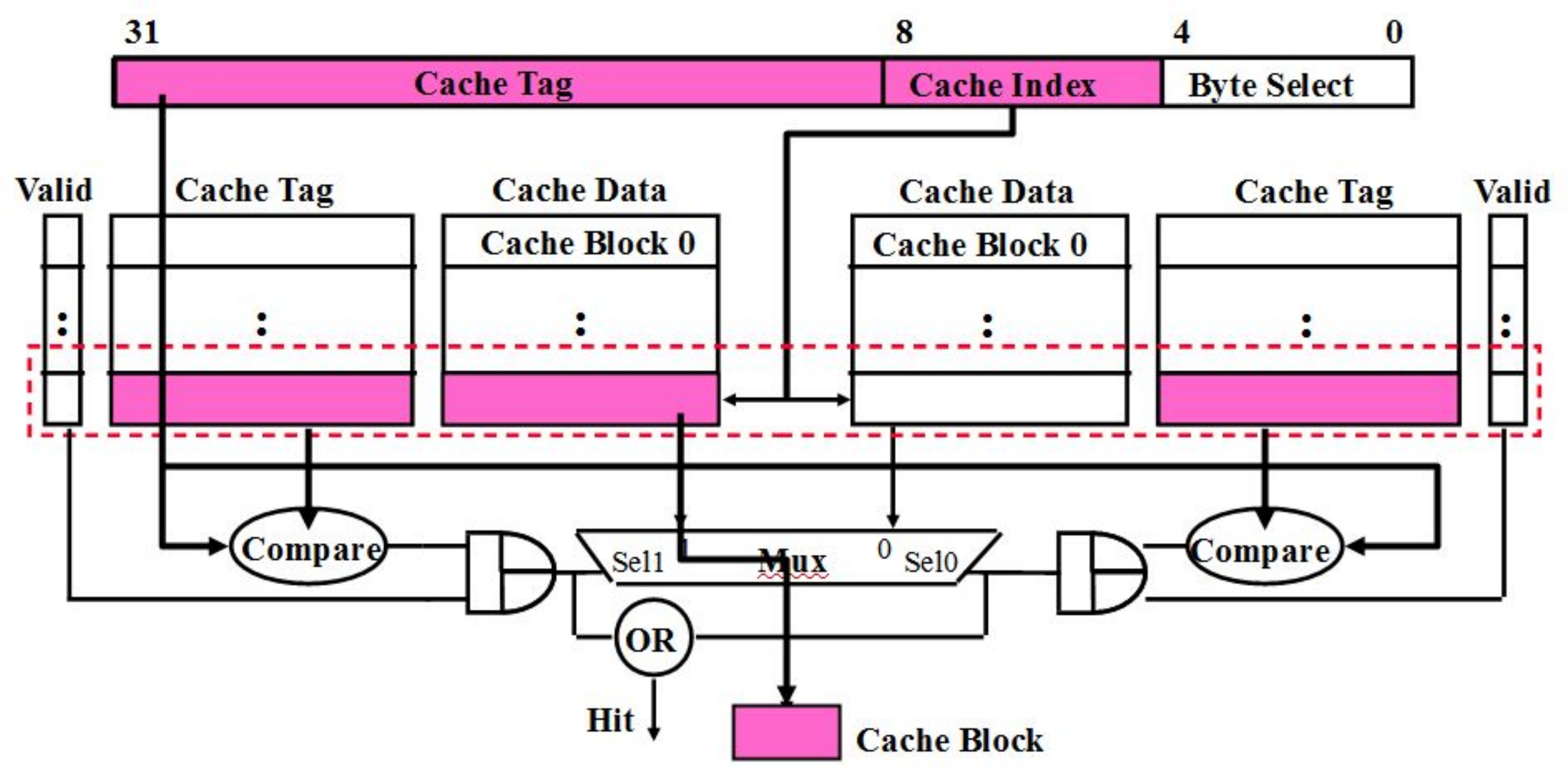
Direct Mapped Cache



Fully Associative Cache



N-Way Set Associative Cache



Cache Misses

- Types of cache misses:
  - Compulsory Miss: Occurs when first accessing a block
  - Capacity Miss: Occurs when accessing a block that was evicted due to the cache being full
  - Conflict Miss: Occurs when accessing a block that was evicted due to a set being full
  - Coherence Miss: Occurs when accessing a block that has invalid data

Fall 2018 MT2 Q6

Find hit rate for N = M = 8, 1B elements

Cache: 4-way, 512B, LRU eviction, 8B blocks

```
for (int j=0; j < N; j++) {
   for (int i=0; i < M; i++) {
     y[i] += A[i][j] * x[i];
     z[i] = i;
   }
}
```

- A is saved as a 2d array starting at the address 0x10
- X is saved as an array starting at address 0x500000010
- Y is saved as an array starting at address 0x1000000010
- Z is saved as an array starting at address 0x1500000010
- The matrix A is stored as a row-major matrix (i.e., rows are stored contiguously in memory)

Fall 2018 MT2 Q6

Find hit rate for N = M = 8, 1B elements

5*8*8 = 320 total accesses. 8*8 loop iterations. 5 memory operations: read A[i][j], x[i], y[i]. Write y[i], z[i].

The matrix and vectors fit inside the cache.

- 1 block for x, y, and z. Brought in and never kicked out.
- 8 blocks for A. Brought in and never kicked out.
- 320 accesses - 8 - 1 - 1 -1 = 309 hits.

HR = 309/320

Eviction

cache/page replacement and write policies

Replacement Policies

1. **FIFO** - throw out the oldest page
2. **RANDOM** - pick a random page for every replacement
3. **MIN** - replace page that won’t be used for longest time
4. **LRU** - replace page that hasn’t been used for longest time
5. **MRU** - replace page that was just used
6. **Clock** - approximates LRU; crude partitioning of pages into two groups, recently used/ NOT recently used
7. **Nth Chance** - give page N chances; a more granular partitioning for Clock algorithm

Question

Can a direct mapped cache sometimes have higher hit rate than a fully associative cache with an LRU replacement policy (with the same access pattern)?

Question

Can a direct mapped cache sometimes have higher hit rate than a fully associative cache with an LRU replacement policy (with the same access pattern)?

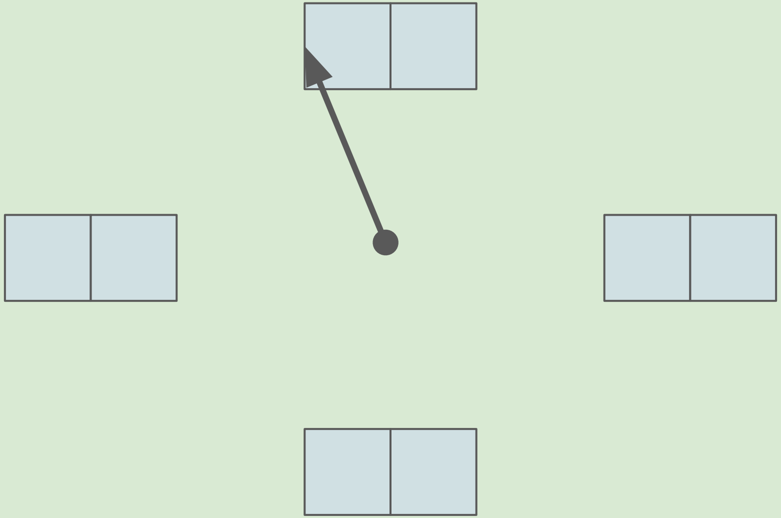
Yes. If a cache has N blocks, then repeatedly accessing N+1 sequential addresses will exhibit a higher hit rate for direct mapped caches.

Clock Algorithm

- Each cache entry (or PTE) keeps track of extra bit called use bit (a.k.a. clock bit, reference bit)
  - Use bit 1 means young; use bit 0 means old
- Upon hit, set the entry's use bit to 1
- Upon miss
  - Clock hand sweeps over entries until finding one to evict:
    - If entry has use bit 1, clear it (set it to 0)
    - If entry has use bit 0, evict this entry
  - Sets use bit of new entry to 1

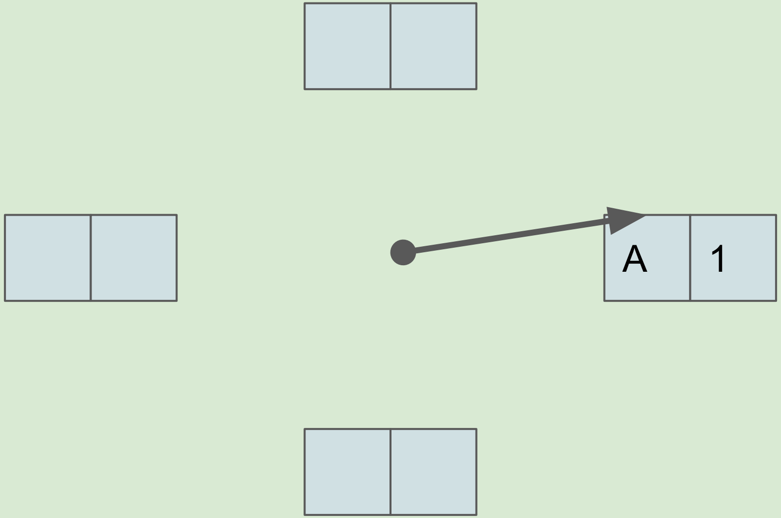
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



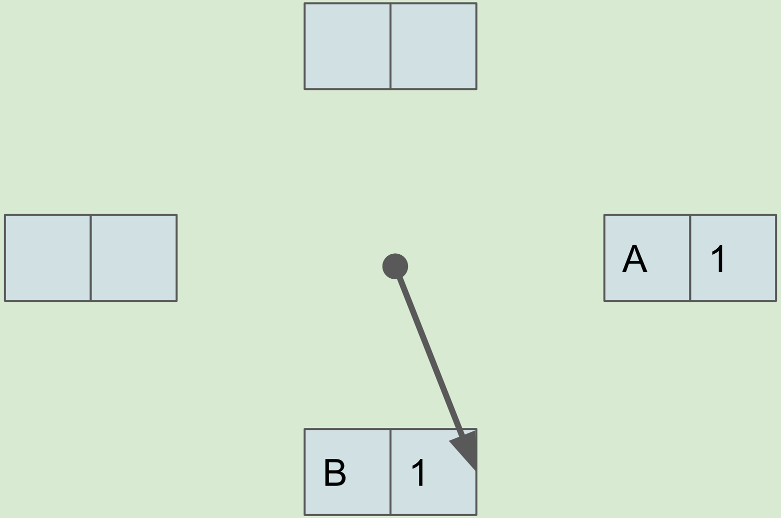
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



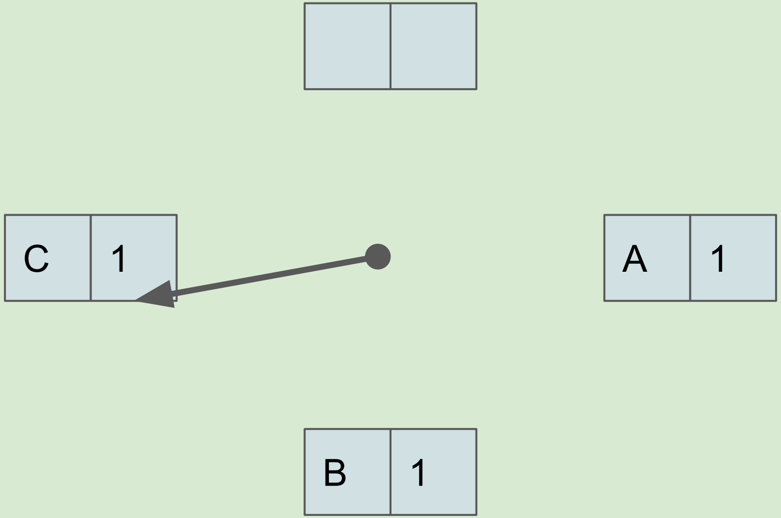
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



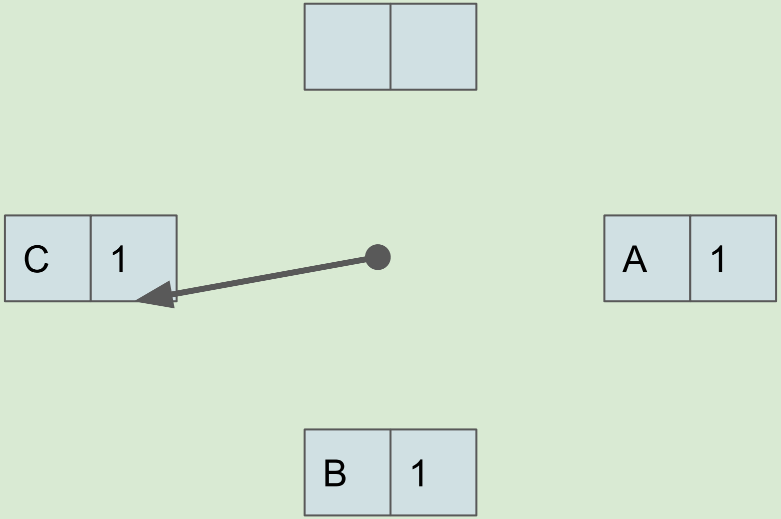
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



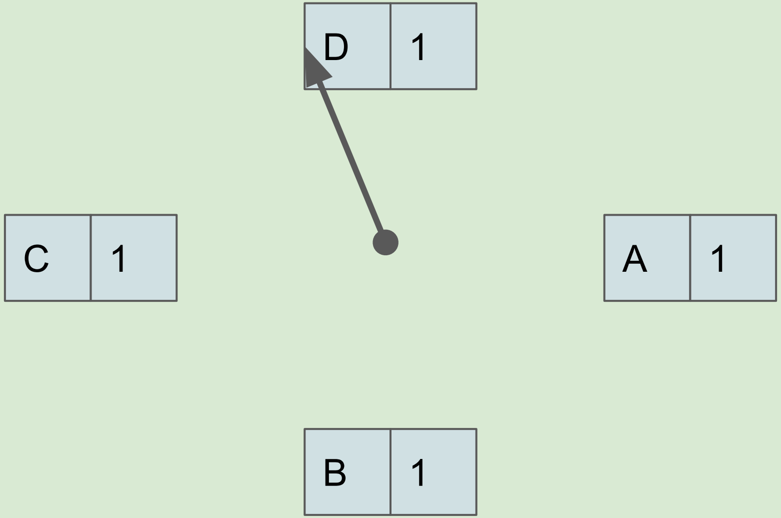
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



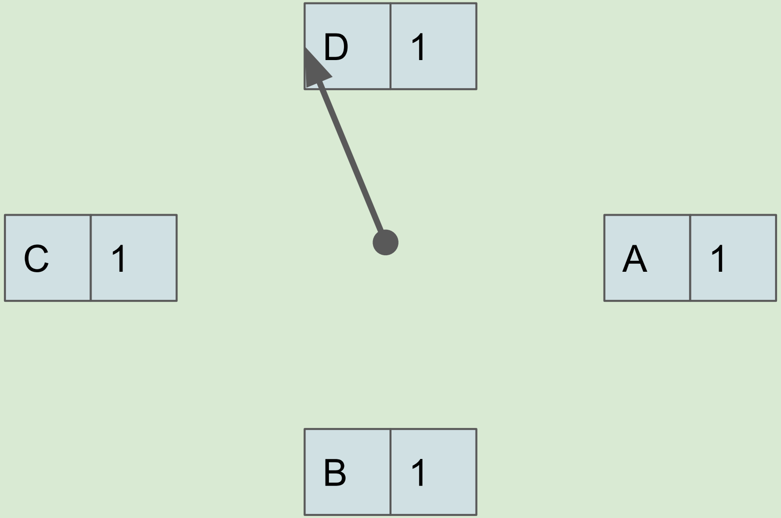
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



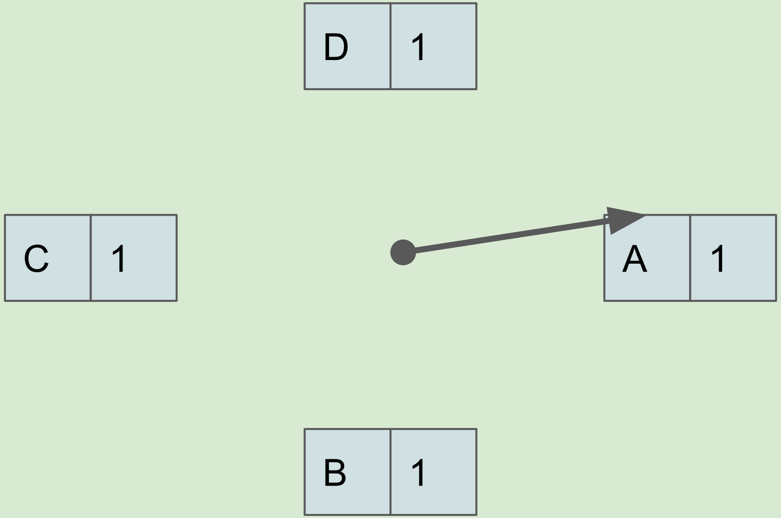
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



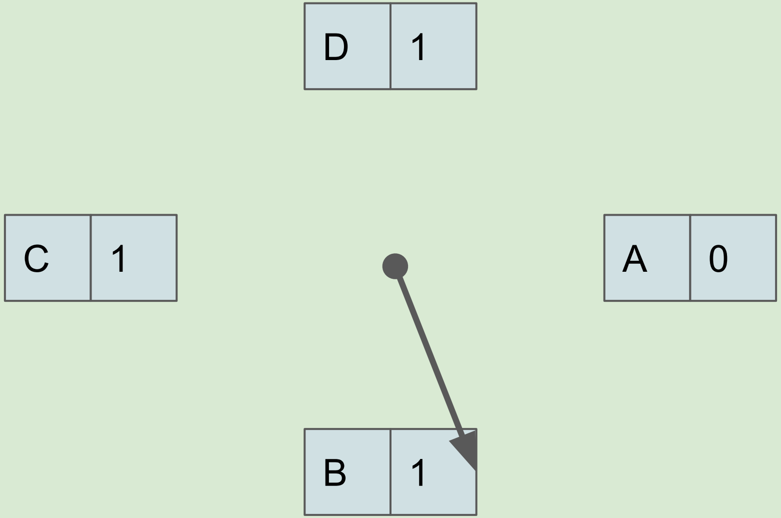
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



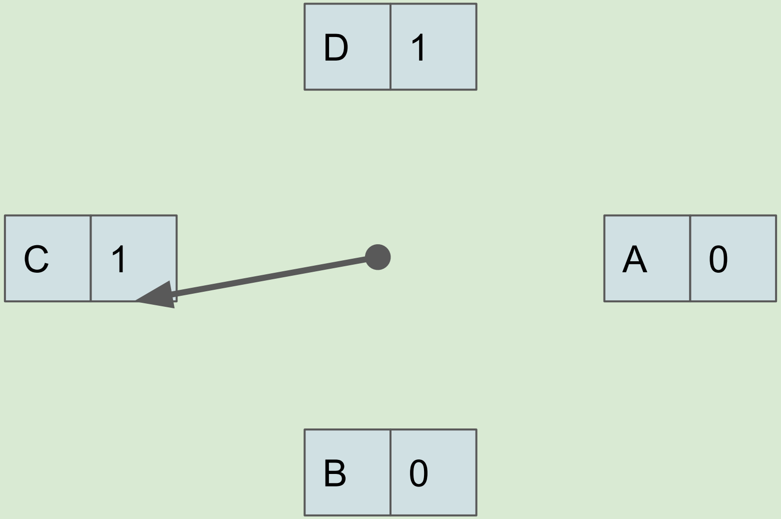
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



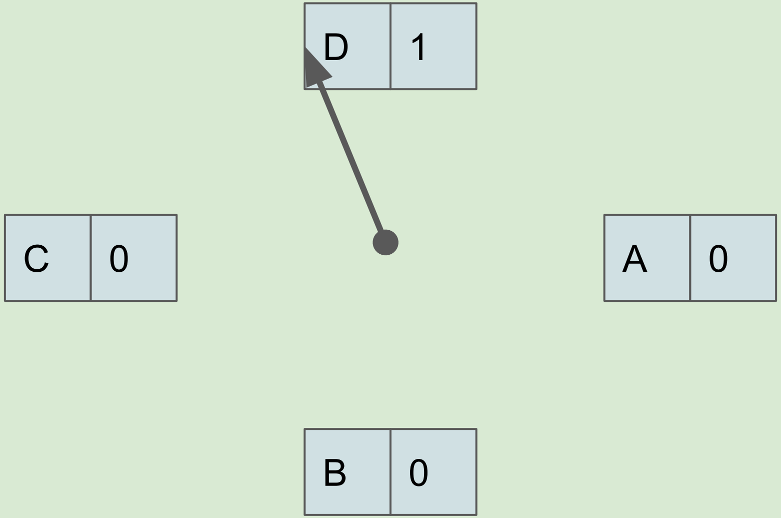
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



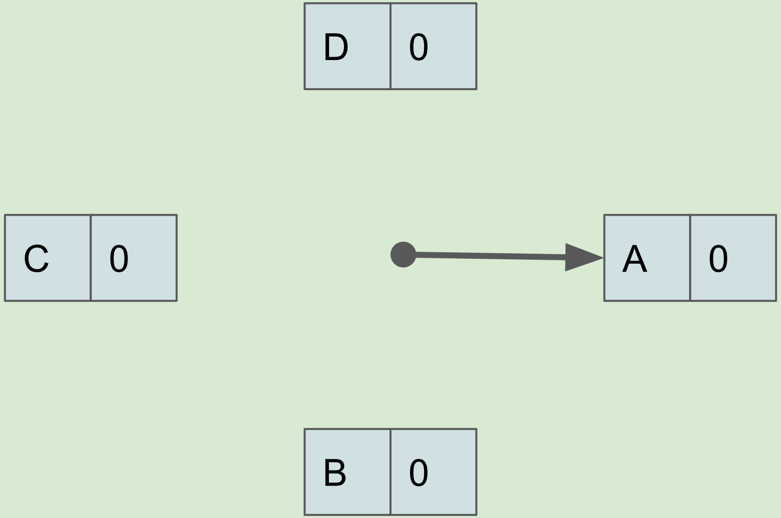
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



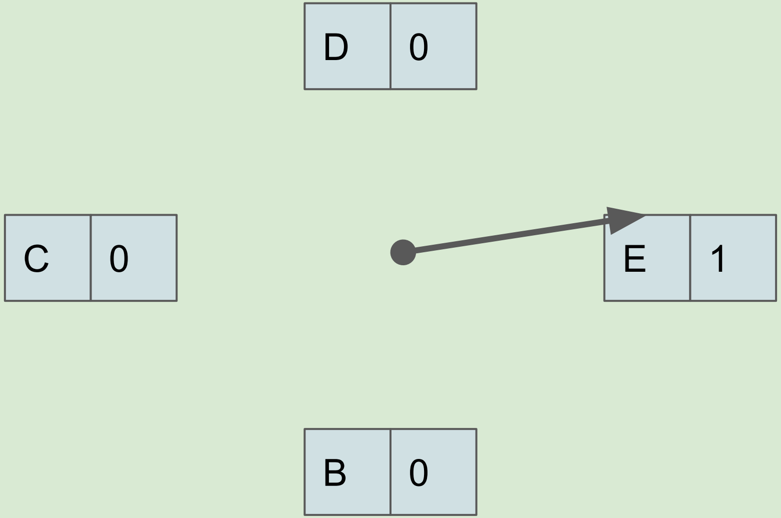
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



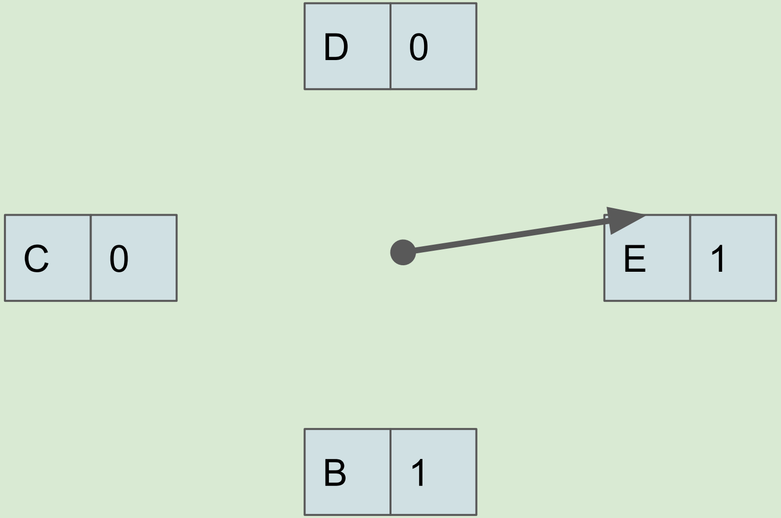
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



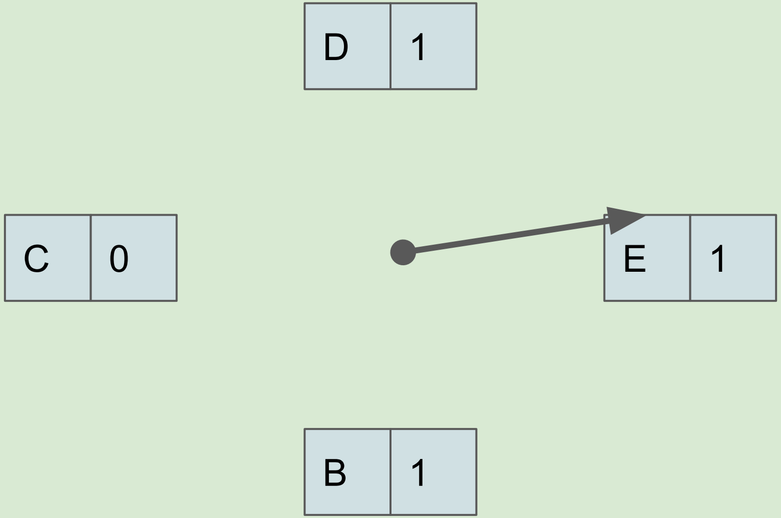
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



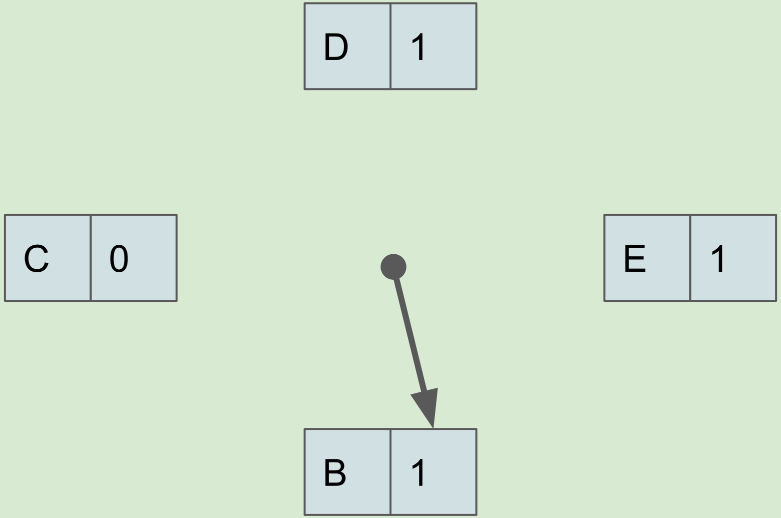
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



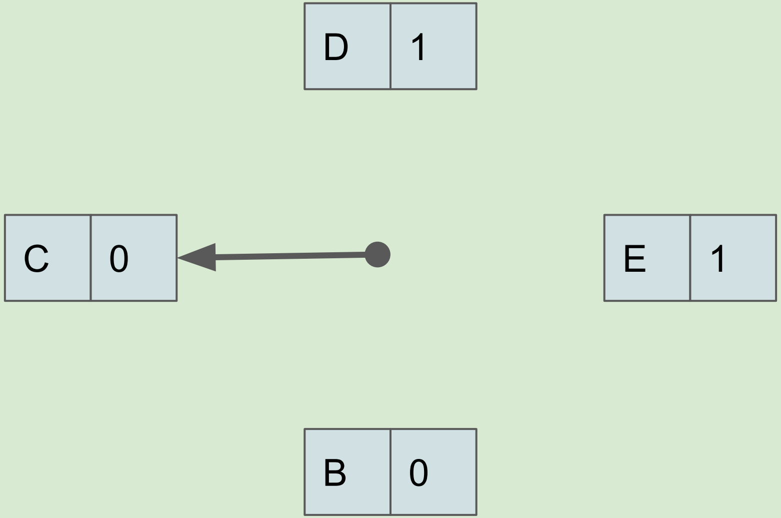
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



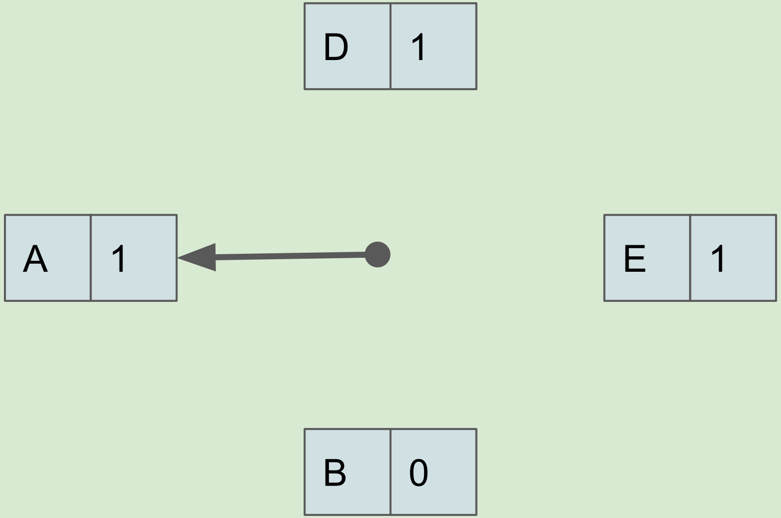
Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



Clock Algorithm Example

We have a 4-entry empty cache/PT and access A B C B D A E B D A



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"

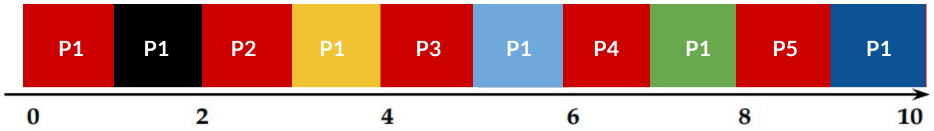
EEVDF: Algorithm

- 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: $\mathbf{V_d^{i+1} = V_e^{i+1} + R/f_i}$

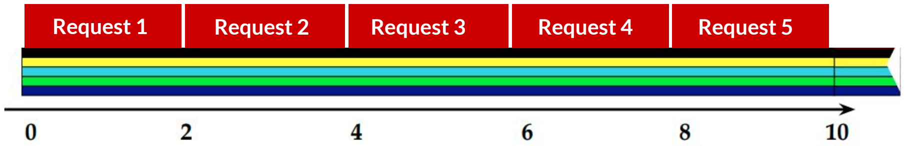
    Rinse and repeat!
  - In words: Your next eligible time gets compensated by the “lag” you experienced.
    - Positive lag pulls the next eligible time closer. Negative lag postpones it further away.
- Observation:
  - Flows with positive lag are always eligible.

Concept Check

EEVDF



Fluid Flow System

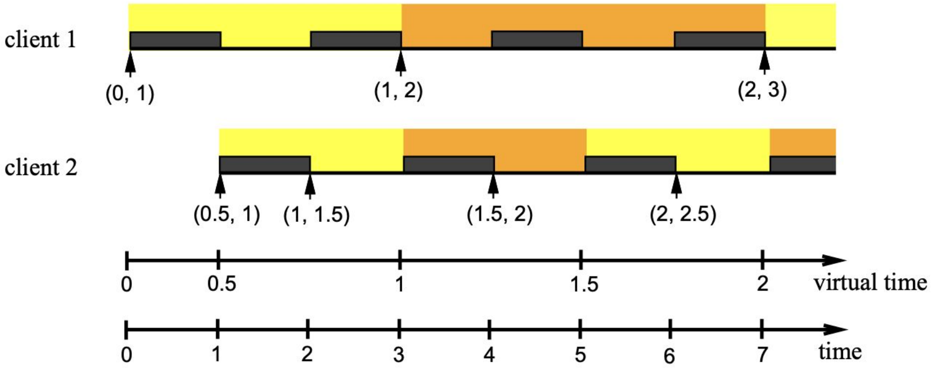


1. From t=0 to 2s, why must red packet 1 be scheduled before any non-red packet? Why does this also apply for t=4 to 6s for red packet 3?

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

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=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 **k^{th} request.**

t_0^i = time that request was initiated.

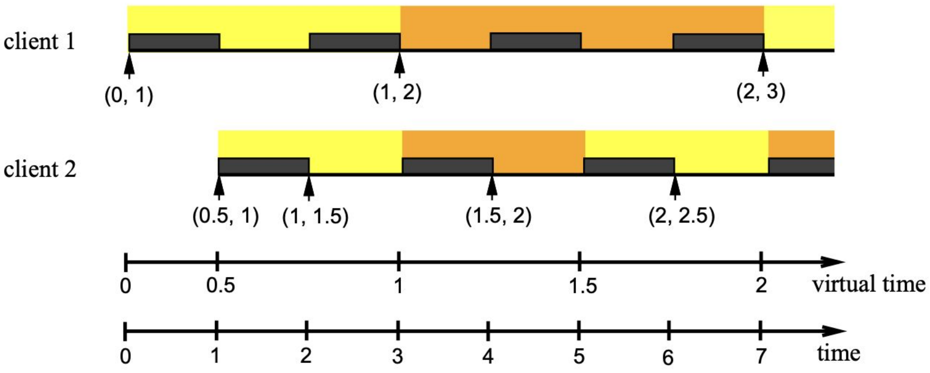
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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	**dV(t)** **--—--** **dt**	Client 1 (C1)	Client 2 (C2)
0	0	**C1**	**0.5**	$\mathbf{v}_e$=0, =$\mathbf{v}_d$=1	—
1	0.5	C1, C2	**0.25**	$\mathbf{v}_e$= 0, $\mathbf{v}_d$= 1	$\mathbf{t}_0^{\,i}$ = 1 $\mathbf{v}_e$ = v(1) = 1/w_1 * 1 = ½ , $\mathbf{v}_d$= 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)*

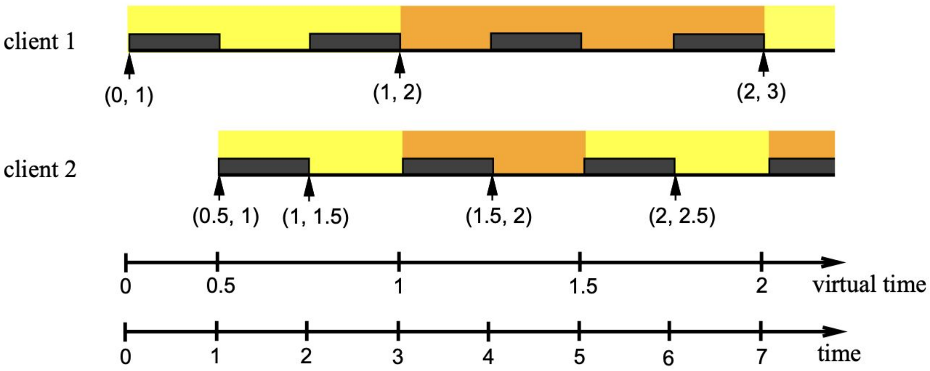
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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	**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)*

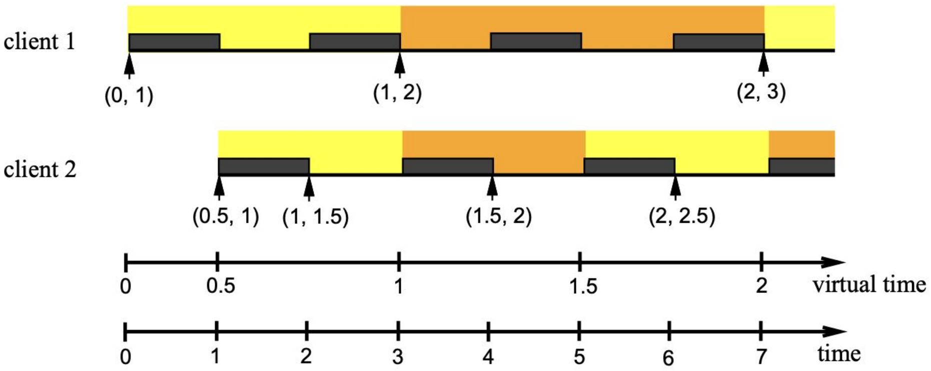
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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

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)*

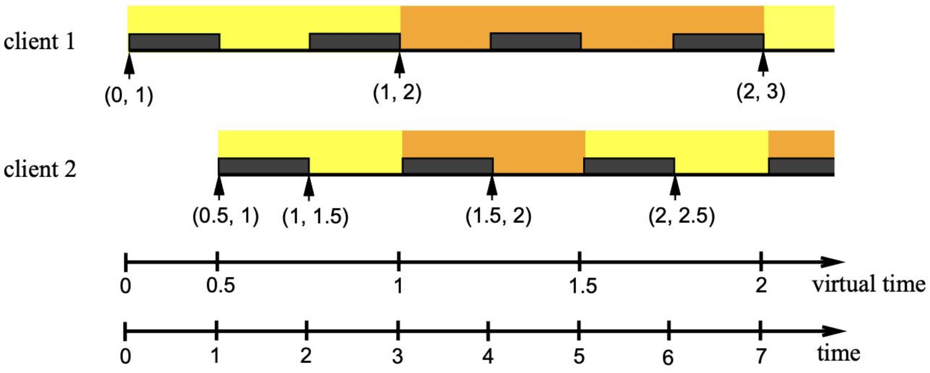
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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



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)*

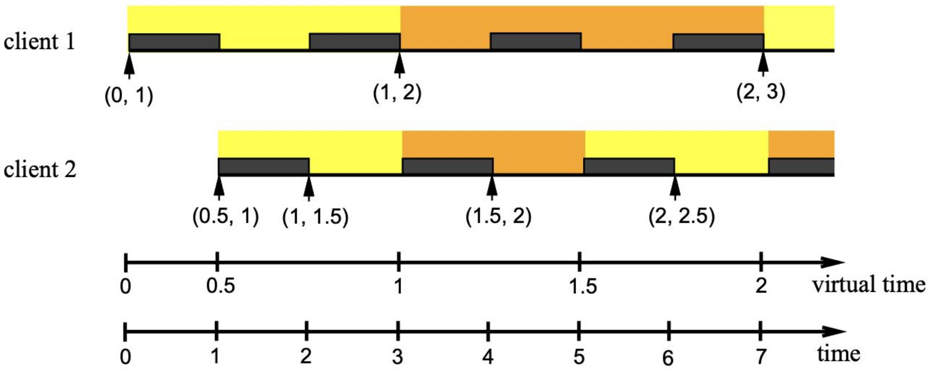
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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	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)*

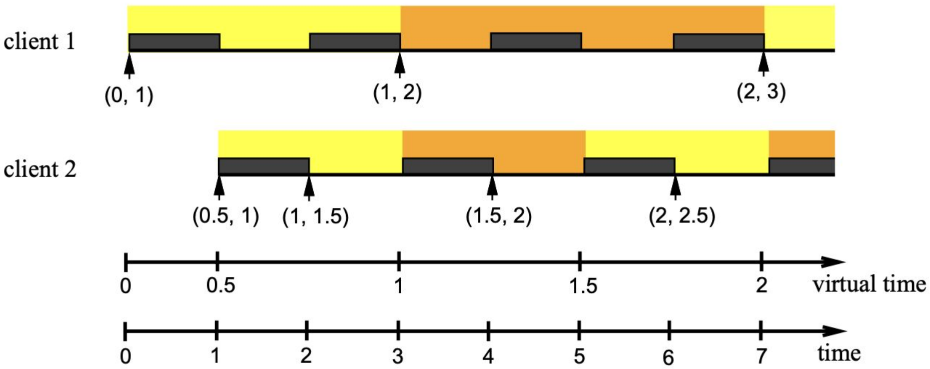
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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)*

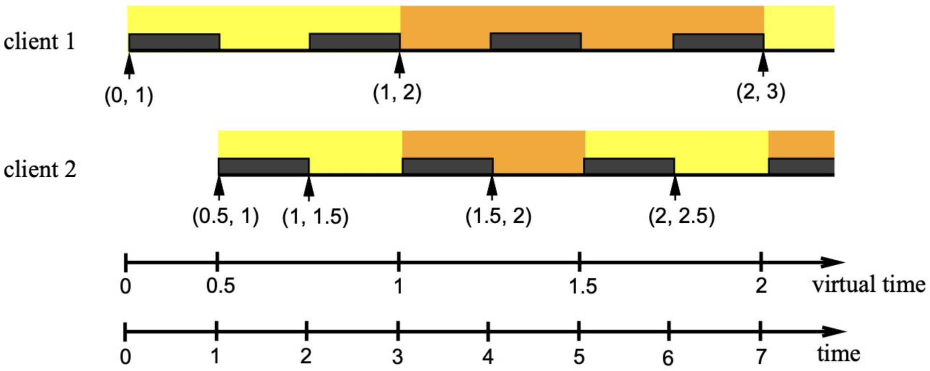
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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	$\frac{dV(t)}{dt}$	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



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

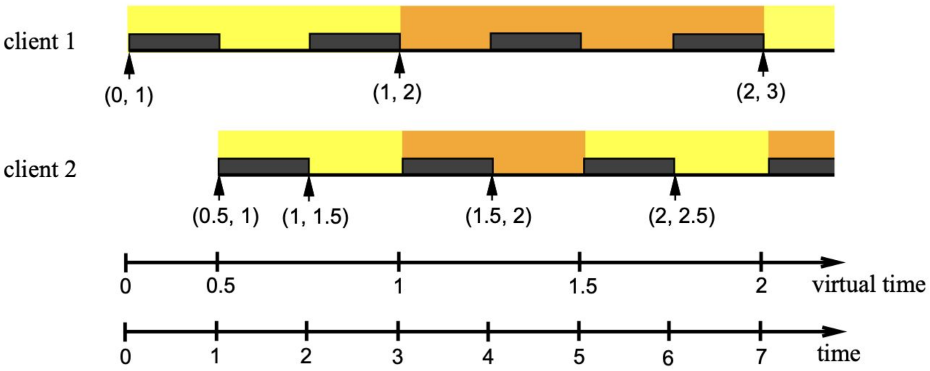
Eq. 2 $$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i},$$

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
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.



<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)*

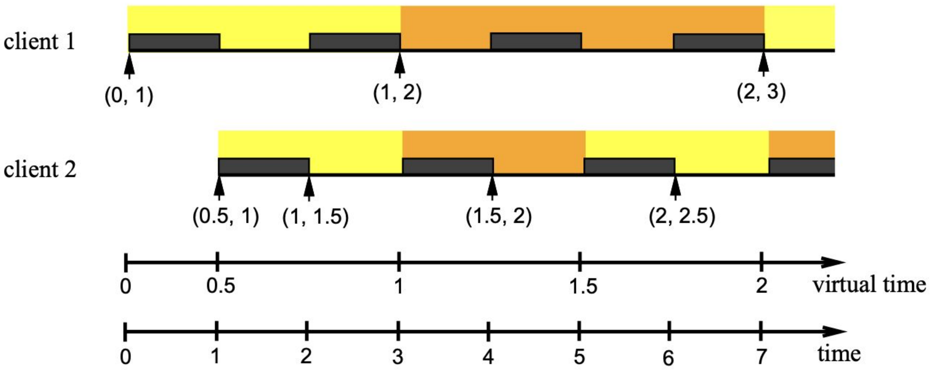
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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	1.50	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)*

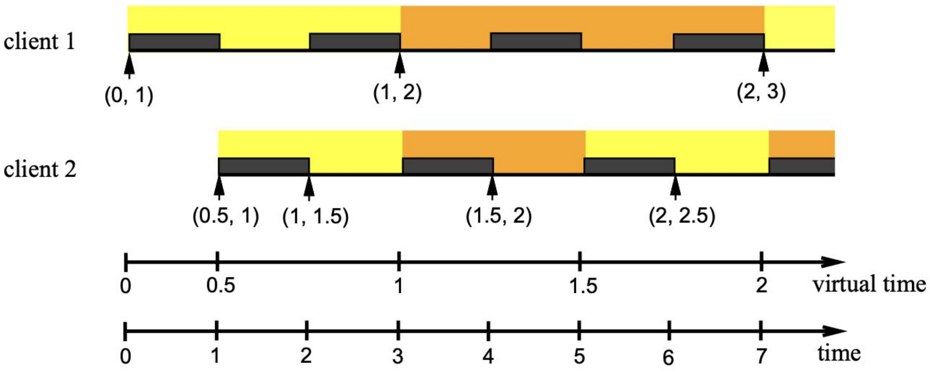
Eq. 1 $$ve^{(1)} = V(t_0^i),$$

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

Practice

Suppose we have the following memory accesses: D C B A D C E D C B A E.

If we have 3 empty physical frames of memory, how many page faults will occur when using FIFO, LRU, or Clock? What if there are 4 frames?

FIFO:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D											
2		C										
3			B									

How many page faults?

Answers

FIFO:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D			A			E					
2		C			D					B		
3			B			C					A	

9 Page Faults

LRU:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D											
2		C										
3			B									

How many page faults?

Answers

LRU:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D			A			E			B		
2		C			D						A	
3			B			C						E

10 Page Faults

Clock:

	D	C	B	A	D	C	E	D	C	B	A	E
1			B									
2	D											
3		C										

How many page faults?

Answers

Clock:

	D	C	B	A	D	C	E	D	C	B	A	E
1			B			C					A	
2	D			A			E					
3		C			D					B		

9 Page Faults

FIFO:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D											
2		C										
3			B									
4				A								

How many page faults?

Answers

FIFO:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D						E				A	
2		C						D				E
3			B						C			
4				A						B		

10 Page Faults

LRU:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D											
2		C										
3			B									
4				A								

How many page faults?

Answers

LRU:

	D	C	B	A	D	C	E	D	C	B	A	E
1	D											E
2		C										
3			B				E				A	
4				A						B		

8 Page Faults

Clock:

	D	C	B	A	D	C	E	D	C	B	A	E
1				A								
2	D											
3		C										
4			B									

How many page faults?

Answers

Clock:

	D	C	B	A	D	C	E	D	C	B	A	E
1				A						B		
2	D						E				A	
3		C						D				E
4			B						C			

10 Page Faults

Conceptual Question

Q: Does adding more entries to a cache/PT reduce miss rate for every replacement policy?

Conceptual Question

Q: Does adding more entries to a cache/PT reduce miss rate for every replacement policy?

No. Known as Bélády's anomaly.

- LRU not affected
- FIFO clock, Nth chance affected

Conceptual Question

Q: Does adding more entries to a cache/PT reduce miss rate for every replacement policy?

No. Counterexample with FIFO accessing A B C D A B E A B C D E

- 3 entries: 9 misses A B C D A B E A B C D E
- 4 entries: 10 misses A B C D A B E A B C D E

Write Policies

- write through: write both cache & memory
  - Simple design
- write back: write cache only, memory is written only when the entry is evicted
  - A dirty bit per entry to denote whether the entry needs to be written back to memory upon eviction

Write Policies Question

- Q: For write through (WT) policy, how many accesses to memory must you make for a
  a. Read hit
  b. Write hit
  c. Read miss with eviction
  d. Write miss with eviction

Write Policies Question

- Q: For write through (WT) policy, how many accesses to memory must you make for a
    a. Read hit
    b. Write hit
    c. Read miss with eviction
    d. Write miss with eviction
- A: 0, 1 (W), 1 (R), 2 (RW)

Write Policies Question

- Q: For write back (WB) policy, how many accesses to memory must you make for a
  a. Read hit
  b. Write hit
  c. Read miss with eviction on entry that’s not dirty
  d. Write miss with eviction on entry that’s not dirty
  e. Read miss with eviction on entry that’s dirty
  f. Write miss with eviction on entry that’s dirty

Write Policies Question

- Q: For write back (WB) policy, how many accesses to memory must you make for a
  a. Read hit
  b. Write hit
  c. Read miss with eviction on entry that’s not dirty
  d. Write miss with eviction on entry that’s not dirty
  e. Read miss with eviction on entry that’s dirty
  f. Write miss with eviction on entry that’s dirty
- A: 0, 0, 1 (R), 1 (R), 2 (WR), 2 (WR)

Good luck!

Stay safe and healthy!