CS222
Operating Systems

Lecture 10
Linux CPU Scheduling: CFS

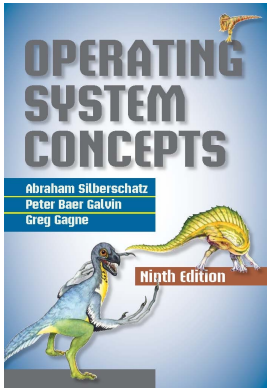
(Section 100001)

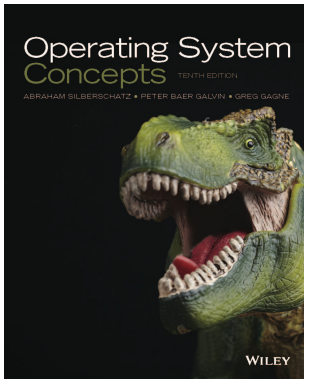
ผศ. ดร. กษิดิศ ชาญเชี่ยว

ckasidit@tu.ac.th

Textbook

- Avi Silberschatz, Peter B. Galvin and Greg Gagne; Operating System Concepts, 9th Edition; John Wiley & Sons, Inc; 2012; ISBN 978-1118063330





- Chapter 5

- Original Slides
- https://www.os-book.com/OS9/slide-dir/index.html
- https://www.os-book.com/OS10/slide-dir/index.html

Operating System Examples

- Linux scheduling
- ~~Windows scheduling~~
- ~~Solaris scheduling~~

Completely Fair Scheduler

- เนื้อหาหลักนำมาจาก presentations จาก Indian Institute of Technology (IIT) ที่ Mudras
  https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
- และจาก Indian Institute of Technology ที่ New Delhi
  https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

ประวัติ Completely Fair Scheduler

- ใช้ใน Linux Kernel version 2.6.23 ในปี 2007 จนถึงปัจจุบัน
- พัฒนาโดย Ingo Molnar โดยนำไอเดียมาจาก the Rotating Staircase Deadline Scheduler (RSDL) โดย Con Kolivas
- ไม่ใช้ Heuristics
- มี Elegant way สำหรับจัดการ I/O-bound และ CPU-bound processes

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

หลักการ Completely Fair Scheduler 1

1. แบ่งเวลาของ CPU ให้ยุติธรรม (Fair) มี N process ก็ได้ 1/N ของ CPU
2. เลือกรัน Process ใน run queue ที่ได้เวลา CPU สะสมน้อยที่สุดก่อน
   (ให้โอกาสคนที่ได้ CPU รวมน้อยก่อน)

Ideal CFS

- สมมุติว่ามี Process ในระบบ N processes
- แต่ละ Process ควรได้รับ 100/N % ของ CPU time
- **กำหนดให้มีช่วงเวลาเรียกว่า Targeted Latency (TL)** เป็นช่วงเวลาหนึ่งที่ Process ทุก Process จะได้ประมวลผล แต่ละ Process ก็จะได้เวลา TL/N CPU time ในการประมวลผล
- **Default TL ใน Linux CFS คือ 20 millisec**
- ตัวอย่าง:
  - สมมุติว่ามี 4 processes และ TL = 4 ms
  - แต่ละ Process มี CPU burst ดังตาราง
    - cpu burst ไม่มีคำสั่ง I/O แทรก

ยุติธรรม (Fair) คืออะไร ?
มี N task แล้วจะแบ่งอะไร
แบ่ง CPU เป็นไปไม่ได้ จะทำ
อย่างไรดี...CFS **สร้างนามธรรม**
หน้าต่างของเวลา TL ขึ้นมา
แล้วสร้างสรรค์**กฎเกณฑ์และกลไก**
เพื่อแบ่งพื้นที่ของเวลานั้นให้คน N คน
ใช้งาน ให้ยุติธรรมที่สุดเท่าที่จะทำได้

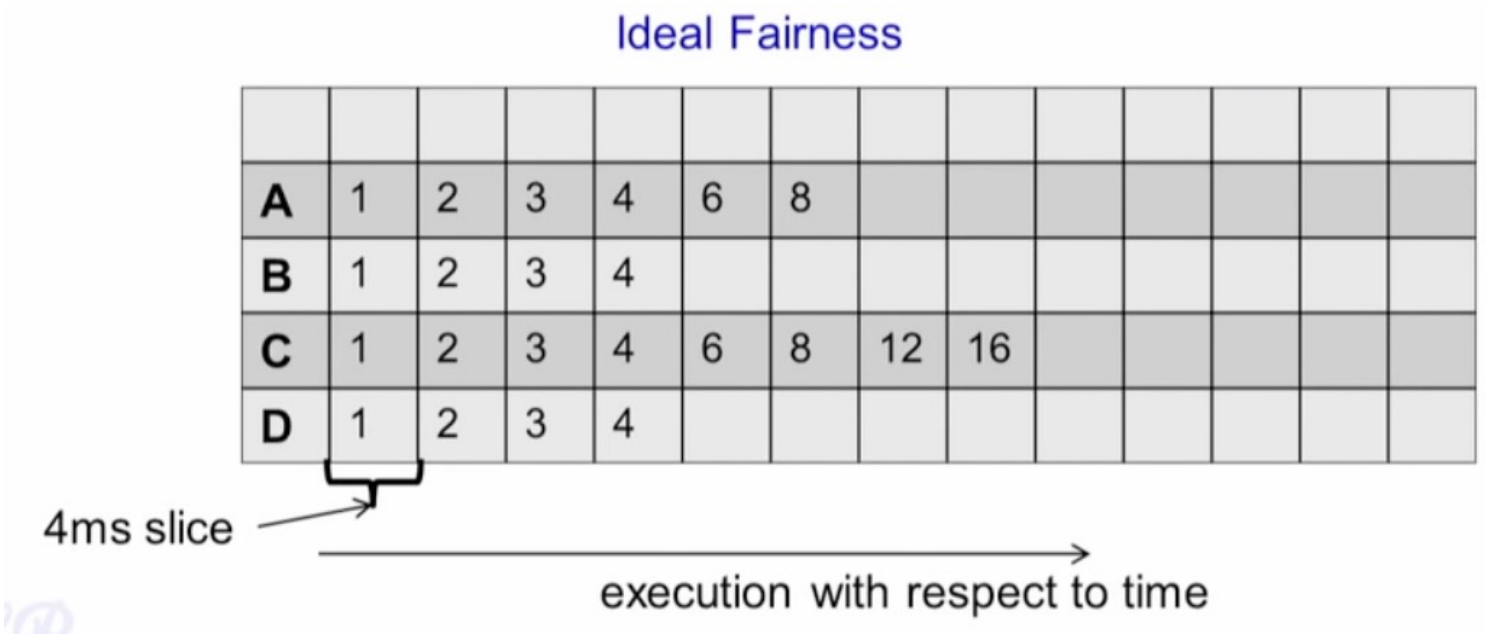
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

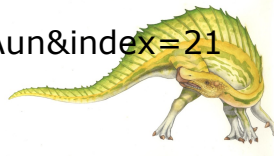
Ideal CFS

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

- สมมุติว่ามี 4 processes และ TL = 4 ms



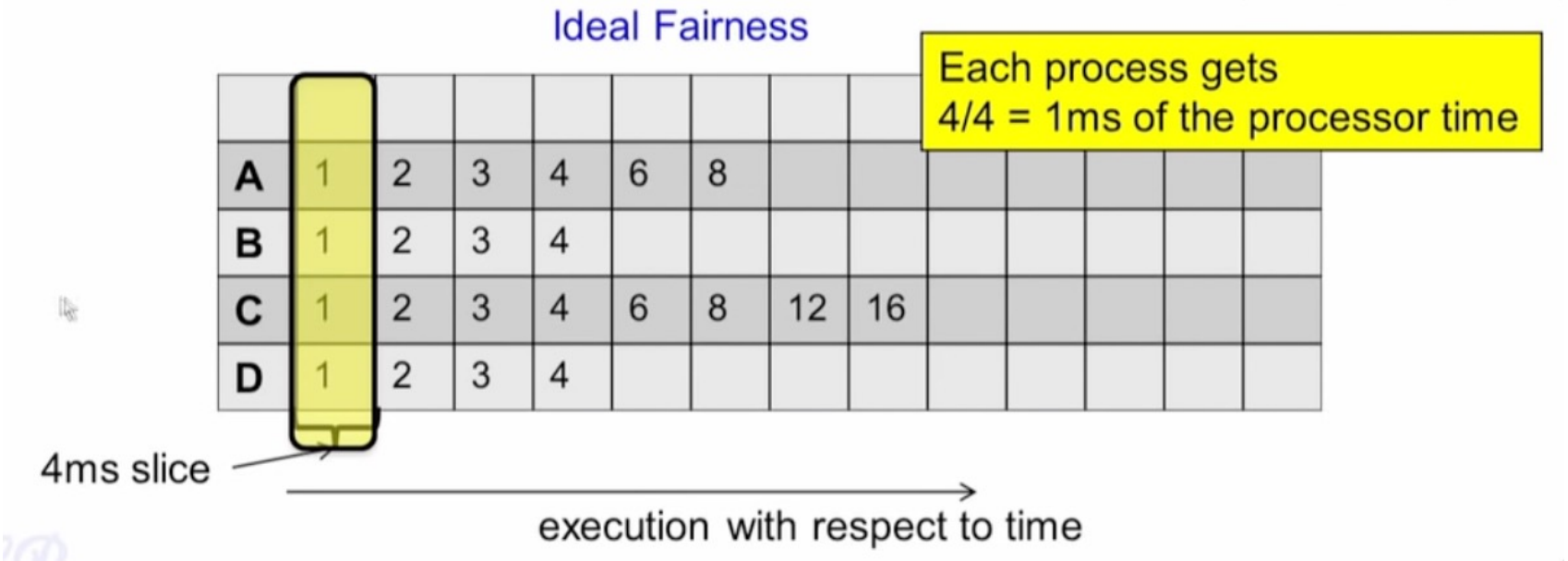
https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf



Ideal CFS

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

- สมมุติว่ามี 4 processes และ TL = 4 ms



https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Ideal CFS

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

- สมมุติว่ามี 4 processes และ TL = 4 ms

Ideal Fairness

Each process gets
4/2 = 2ms of the processor time

A	1	2	3	4	6	8						
B	1	2	3	4								
C	1	2	3	4	6	8	12	16				
D	1	2	3	4								

4ms slice

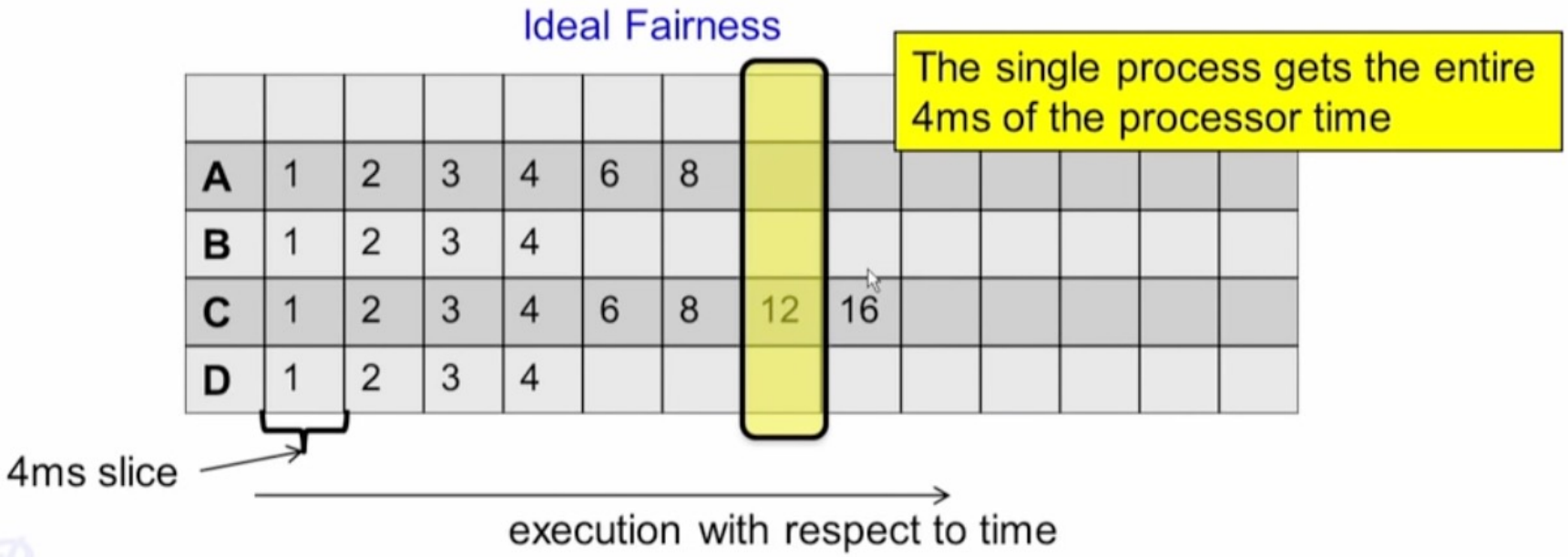
execution with respect to time

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Ideal CFS

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

- สมมุติว่ามี 4 processes และ TL = 4 ms



https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Target Latency

- เวลาของ CPU จะถูกมองเป็นนามธรรมหน้าต่างของเวลาที่จะรัน Task **เรียกว่า Targeted Latency (TL)** ซึ่ง Default TL ของ Linux คือ 20 ms
- ดังนั้นถ้ามี N Tasks **เวลาที่แต่ละ Process จะได้รับ** คือ $\frac{TL}{N}$
- แต่ CFS กำหนดว่า แต่ละ Task ต้องได้รับเวลา **Minimum Granularity (MG)** (Default MG ของ Linux คือ 4 ms)

  ถ้า $\frac{TL}{N} < MG$ **เวลาที่แต่ละ Process จะได้รับ** ก็จะเท่ากับ MG

- ตัวอย่าง

TL = 20 ms และ MG = 4 ms ถ้า N = 4, $\frac{TL}{N} = 5$ มากกว่า MG แต่ละ Task จะได้ 5 ms

แต่ถ้า N = 10 $\frac{TL}{N} = 2$ ซึ่งน้อยกว่า MG ดังนั้น **แต่ละ Task จะได้เวลา 4 ms** และ เวลารวมที่ใช้รัน Task ทั้งหมด **จะเท่ากับ N x MG = 10 x 4 = 40 ms**

https://opensource.com/article/19/2/fair-scheduling-linux

Target Latency

ข้อสังเกตุ:

1. ในกรณีที่ $\frac{TL}{N} \geq MG$ ค่า Time Slice (TS) เท่ากับ $Ts = \frac{TL}{N}$ ดังนั้นเวลารวมที่ใช้ประมวลผล N Tasks คือ

$$\mathrm{Tf} = \mathrm{N} \times Ts \text{ ซึ่งเท่ากับ } TL$$

2. ในกรณีที่ $\frac{TL}{N} < MG$ ค่า Time Slice (TS) เท่ากับ $Ts = MG$ ดังนั้นเวลารวมที่ใช้ประมวลผล N Tasks คือ

$$\mathrm{Tv} = \mathrm{N} \times MG \text{ ซึ่งมากกว่า } TL$$

3. ถ้ามองอีกมุมหนึ่งว่า เวลาที่ใช้ประมวลผล N Tasks คือ General Target Latency เราก็จะมี General Target Latency สองแบบ แบบแรกคือ Tf ซึ่งเป็นค่าคงที่ (fixed) และแบบที่สองคือ Tv ซึ่งเปลี่ยนขนาดได้ (variable) ตามค่า N
4. **ทั้ง Tf และ Tv มีความ Fair ทั้งคู่เพราะทั้งสองให้เวลาทุก Task เท่ากัน**

https://opensource.com/article/19/2/fair-scheduling-linux

แนะนำ vruntime

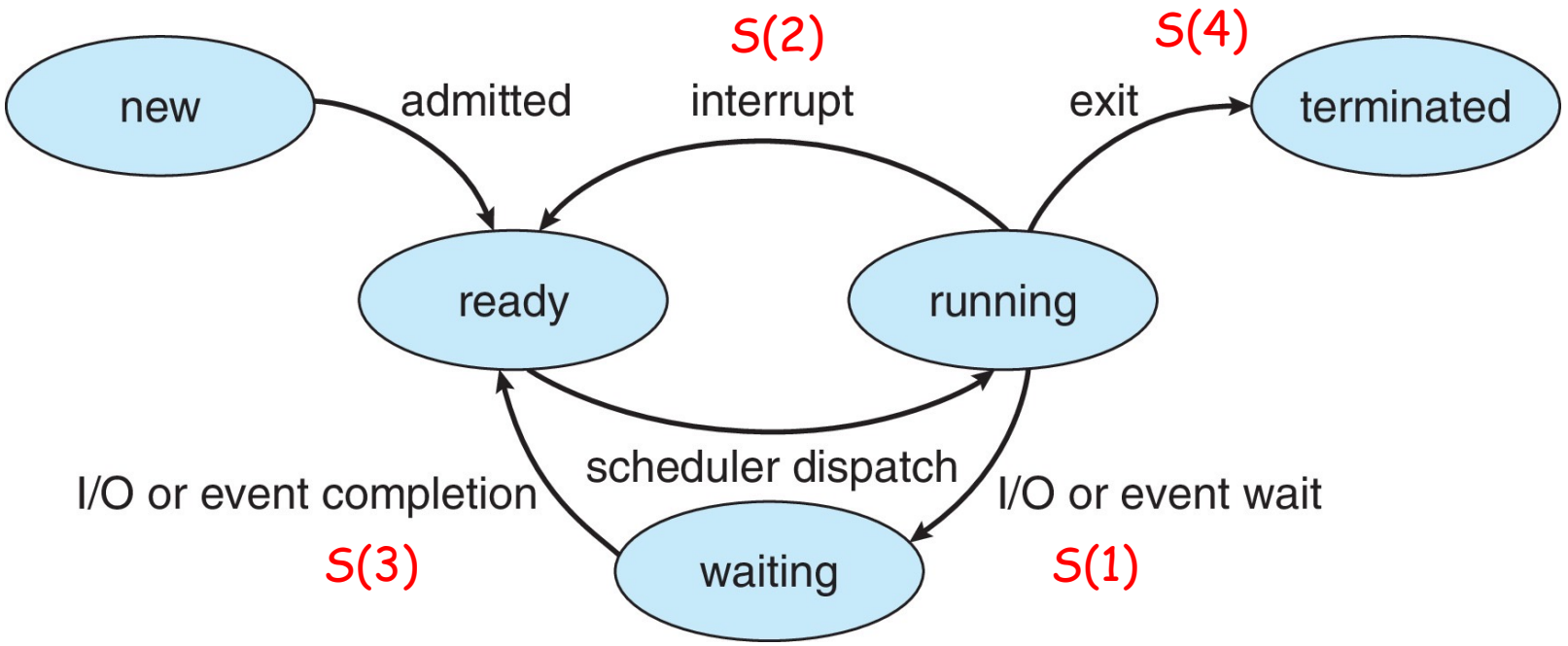
- **วัตถุประสงค์หลัก:** CFS จะใช้ vruntime **เป็น Index สำหรับเปรียบเทียบ task ใน run queue** ว่าใครใช้ CPU น้อยที่สุดและควรได้รับเลือกเข้ามาประมวลผล CPU
- ใน PCB ของแต่ละ Task จะมี ตัวแปรชื่อ virtual runtime (vruntime)
- **vruntime คือนามธรรมของเวลาเสมือนสะสมที่ CFS กำหนดให้แต่ละ Task (ตามกฎที่จะได้กล่าวถึงต่อไป)**
- Slide ถัดไปจะแสดงว่า CFS scheduler จะเลือก task ใน run queue เข้าใช้ CPU และปรับ vruntime ตอนไหนบ้าง

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

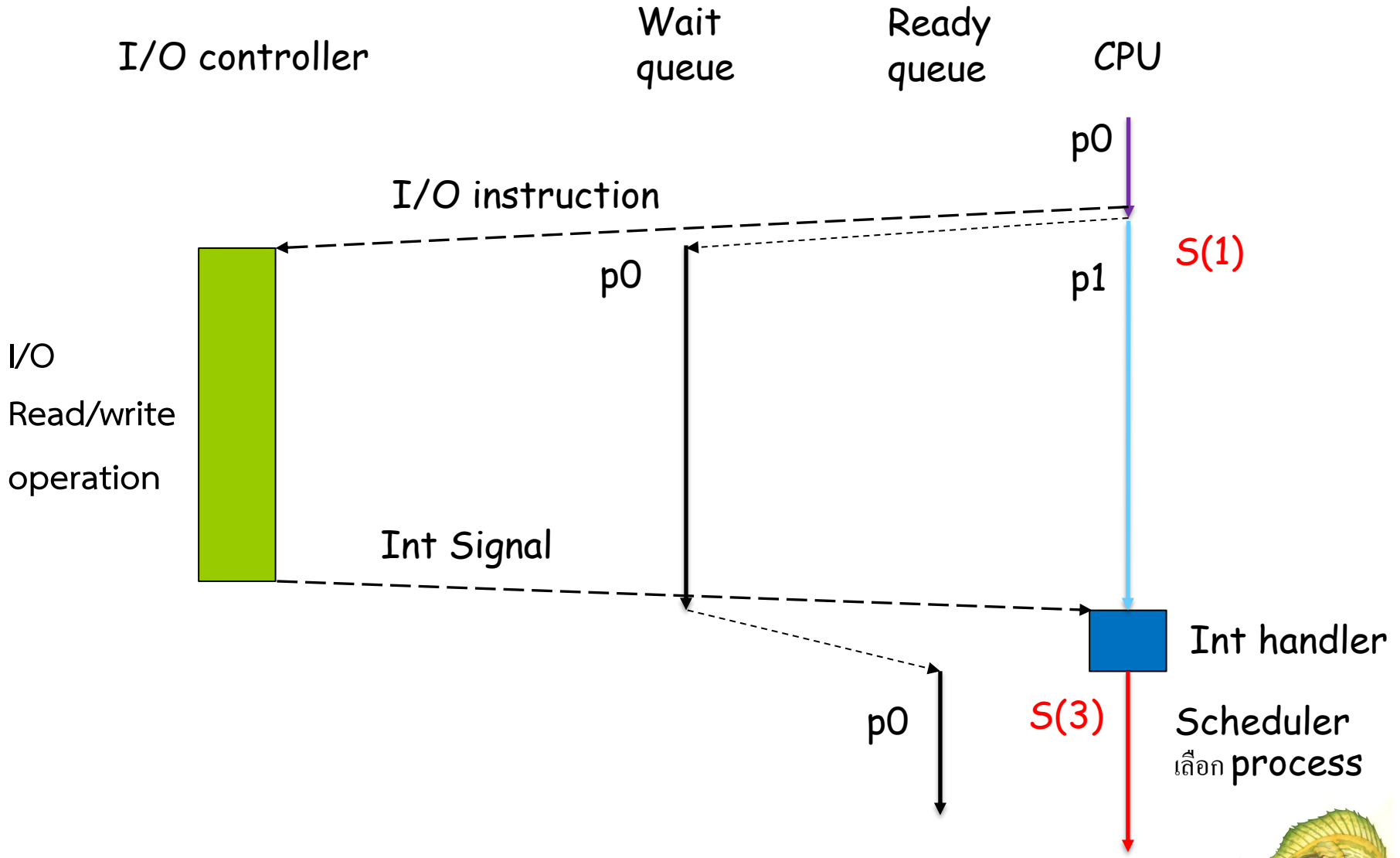
ทบทวน: CPU Scheduling เกิดเมื่อ....

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state S(1)
  2. Switches from running to ready state S(2)
  3. Switches from waiting to ready S(3)
  4. Terminates S(4)
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.
- **Linux CFS จะพิจารณา เลือก Process ใหม่จาก run queue ในทุกสถานการณ์ s(1) s(2) s(3) และ s(4)**

ทบทวน: Diagram of Process State



I/O read/write (s3 preemption)



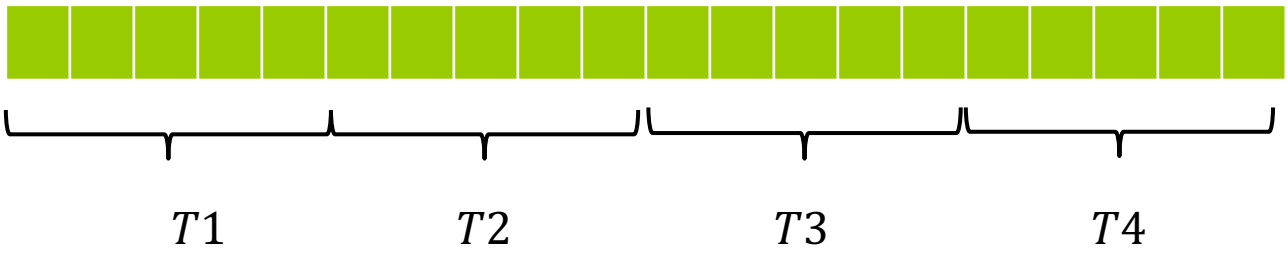
Time slice

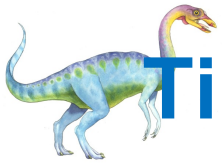
- CFS กำหนดค่า Time Slice ให้กับแต่ละ Task ตาม weight (ซึ่งคำนวณจากค่า static priority ของ task ที่กำหนดตอนรัน task นั้น)
- ถ้า priority สูง (ค่า nice หรือ priority level น้อย) weight จะมาก
- ถ้า priority ต่ำ (ค่า nice หรือ priority level มาก) weight จะน้อย
- ตอนที่ Task ถูกเลือกเข้าใช้งาน CPU CFS จะนำค่า weight มาคำนวณ และกำหนดค่า Time Slice ให้กับ Task นั้น

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Time Slice (weight หรือ priority เท่ากัน)

- ใน CFS ค่า TL อาจเป็น Tf หรือ Tv ก็ได้ สมมุติว่า $TL = \boldsymbol{Tx}$
- สมมุติว่ามี N task ตามปกติแล้ว ทุก task จะได้เวลา
- Time slice คือ $ts = \frac{Tx}{N}$
- สมมุติว่า Tx = 20 ms และ N = 4 เราได้ Ts = 5 ms
- ทุก task มี weight เท่ากัน คือ 1/n = ¼ = 0.25
- รวมกันทุกสัดส่วน ¼ + ¼ + ¼ + ¼ = 1





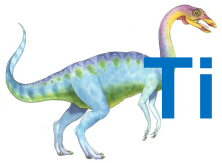
Time Slice (weight หรือ priority ไม่เท่ากัน)

- สมมุติว่ามี N task มี weight คือ w1, w2, w3, .. wN และให้ $tw = \sum_{i=1}^{n} w_i$
- แต่ละ task จะได้เวลา Time Slice คือ
- สัดส่วน weight ของแต่ละ task จะเท่ากับ $\frac{w_1}{tw}, \frac{w_2}{tw}, \ldots, \frac{w_n}{tw}$

รวมกันเท่ากับ 1

- Time slice ของ task $\boldsymbol{i}$ เท่ากับ

$$ts_i = \frac{w_i}{tw} \times T$$



Time Slice (weight หรือ priority ไม่เท่ากัน)

- สมมุติว่ามี N task มี weight คือ w1, w2, w3, w4 คือ
- 2, 8, 6, 4 ดังนั้น $tw = \sum_{i=1}^{n} w_i = 20$
- แต่ละ task จะได้เวลา Time Slice คือ 2, 8, 6, 4
- สัดส่วน weight ของแต่ละ task จะเท่ากับ $\underbrace{\frac{2}{20}, \frac{8}{20}, \frac{6}{20}, \frac{4}{20}}$ รวมกันเท่ากับ 1
- Time slice ของ task 1 เท่ากับ

$$ts_1 = \frac{w_i}{tw} \times Tx = \left(\frac{2}{20}\right) * 20 = 2$$ เป็นต้น

vruntime

- ใน PCB ของแต่ละ Task จะมี ตัวแปรชื่อ virtual runtime (vruntime)
- vruntime คือนามธรรมของเวลาเสมือน**สะสม**ที่ CFS กำหนดให้แต่ละ Task (ตามกฎที่จะได้กล่าวถึงต่อไป)
- **สมมุติว่าทุก Task มี weight เท่ากัน และมี Time Slice = Ts ถ้า Task ที่ถูกขัดจังหวะใช้งานซีพียูไป t time unit (nanoseconds)**
  - **vruntime += t (หรือ vruntime = vruntime + t)**
- vruntime เป็นค่าเวลาเสมือนสะสม ที่จะเพิ่มขึ้นเมื่อ task ได้ใช้ CPU
- **วัตถุประสงค์หลัก**ของมันคือ CFS จะใช้มัน**เป็นตัวเปรียบเทียบระหว่าง task ใน run queue** ว่าใครเคยใช้ CPU น้อยที่สุดและสมควรได้รับเลือกให้ใช้ CPU

https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Algorithm ในการเลือก Task

- ถ้า Task Tk ถูกขัดจังหวะและรันไป t จาก Time Slice Ts
- ในกรณีที่ทุก task มี weight หรือ priority เท่ากัน
- CFS จะเพิ่มค่า vruntime += t
- CFS จะเปรียบเทียบ vruntime ของทุก task ใน run queue รวมทั้งของ Tk และเลือก task ที่มี vruntime น้อยที่สุด
- ถ้า T1kยังคงน้อยกว่าทุก task ใน runqueue T1kก็ใช้ cpu ต่อ
- ถ้ามีคนที่น้อยกว่า Tk CFS ก็จะเลือกมา แล้วเอา Tk ใส่ใน run queue

Time Slice (weight หรือ priority เท่ากัน)

- สมมุติว่า Tx = 20 ms และ N = 4 เราได้ Ts = 5 ms
- ทุก task มี weight เท่ากัน คือ 1/n = ¼ = 0.25
- ถ้า T1 รันได้ 2 ms จาก 5 ms แล้วถูกขัดจังหวะ ค่า t = 2
- CFS จะเพิ่มค่า vruntime += 2
- CFS จะเปรียบเทียบ vruntime ของทุก task ใน run queue รวมทั้งของ T1 และเลือก task ที่มี vruntime น้อยที่สุด
- ถ้า T1 ยังคงน้อยกว่าทุก task ใน runqueue T1 ก็ใช้ cpu ต่อ
- ถ้ามีคนที่น้อยกว่า T1 CFS ก็จะเลือกมา แล้วเอา T1 ใส่ใน run queue

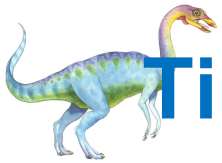
Algorithm ในการเลือก Task (weight ไม่เท่ากัน)

- ถ้า Task Tk ถูกขัดจังหวะและรันไป t จาก Time Slice Ts
- ในกรณีที่ทุก task Tk มี weight คือ w จะได้ Ts ตาม weight
- CFS จะ Normalize ค่าของ t

$$t' = Norm(t, Ts)$$

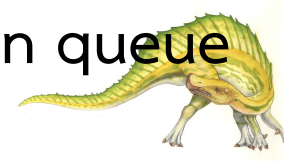
เราจจะอธิบายการ normalize นี้ภายหลัง

- CFS จะเพิ่มค่า vruntime += t'
- CFS จะเปรียบเทียบ vruntime ของทุก task ใน run queue รวมทั้งของ Tk และเลือก task ที่มี vruntime น้อยที่สุด
- ถ้า T1kยังคงน้อยกว่าทุก task ใน runqueue T1kก็ใช้ cpu ต่อ
- ถ้ามีคนที่น้อยกว่า Tk CFS ก็จะเลือกมา แล้วเอา Tk ใส่ใน run queue

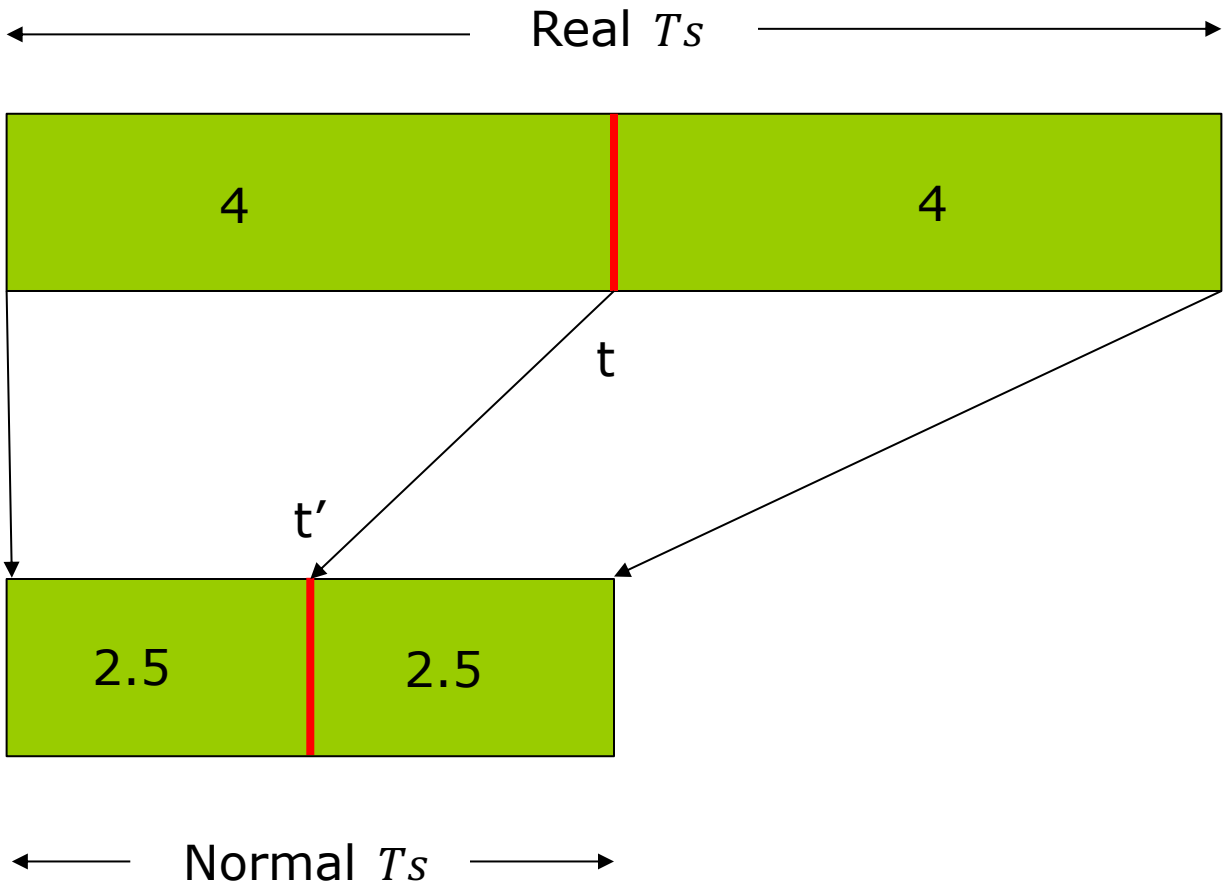


Time Slice (weight หรือ priority ไม่เท่ากัน)

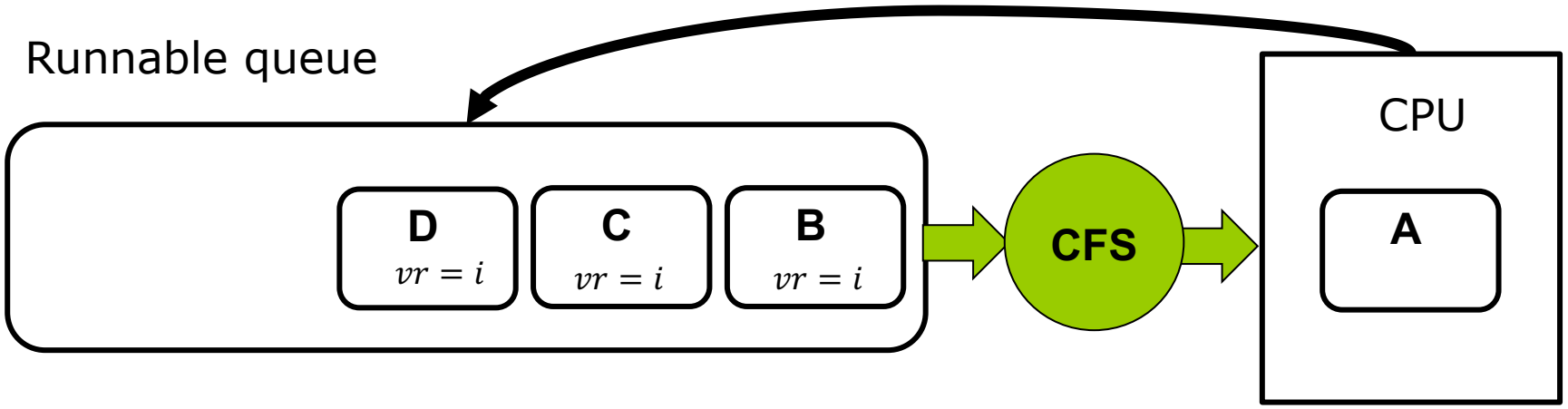
- สมมุติว่ามี N task มี weight คือ w1, w2, w3, w4 คือ
- 2, 8, 6, 4 ดังนั้น $tw = \sum_{i=1}^{n} w_i = 20$
- task จะได้เวลา Time Slice คือ 2, 8, 6, 4
- ถ้า TL = 20 ดังนั้น T2 ได้ Ts = 8
- ถ้า T2 รันได้ 4 ms จาก 8 ms แล้วถูกขัดจังหวะ ค่า t = 4
- CFS จะ normalize t ซึ่งจะได้ t’ = norm(4, 8) =4/8*5= 2.5
- CFS จะเพิ่มค่า vruntime += 2.5
- CFS จะเปรียบเทียบ vruntime ของทุก task ใน run queue รวมทั้งของ T1 และเลือก task ที่มี vruntime น้อยที่สุด
- ถ้า T1 ยังคงน้อยกว่าทุก task ใน runqueue T1 ก็ใช้ cpu ต่อ
- ถ้ามีคนที่น้อยกว่า T1 CFS ก็จะเลือกมา แล้วเอา T1 ใส่ใน run queue



normalization



Run queue and CPU



1. ขั้นตอน CFS (Time Slice Expires)

- สมมุติว่า Task i กำลังประมวลผลอยู่ใน CPU เมื่อ Time Slice ของ Task i หมด CPU จะได้รับสัญญาณ Timer Interrupt ซึ่งจะทำให้ CPU ประมวลผล Interrupt Handler
- Interrupt Handler จะ Preempt (ขัดจังหวะการประมวลผล) Task i แล้ว รัน CFS เพื่อคำนวณค่า vruntime ของ Task i ดังนี้

$$vruntime \mathrel{+}= ts_i$$

โดยที่ ts_i คือค่า Time Slice ที่ผ่านมาของ Task i

- CFS จะทำ Context Switching และนำ Task i ไปไว้ที่ run queue
- CFS จะเลือก task ที่มีค่า vruntime น้อยที่สุด จาก run queue สมมุติว่าชื่อ Task j

1. ขั้นตอน CFS (Time Slice Expires)

- CFS คำนวณเวลา Time Slice ของ Task j ดังนี้
  - สมมุติว่า ts_j คือค่า Time Slice ของ Task j N คือจำนวน Tasks TL คือ Target Latency และ MG คือ Minimal granularity
  - สูตรสำหรับกำหนดค่า Time Slice ของ Task j คือ

$$ts_j = \frac{TL}{N} \text{ ถ้า } \frac{TL}{N} \geq MG$$

หรือ

$$ts_j = MG \text{ ถ้า } \frac{TL}{N} < MG$$

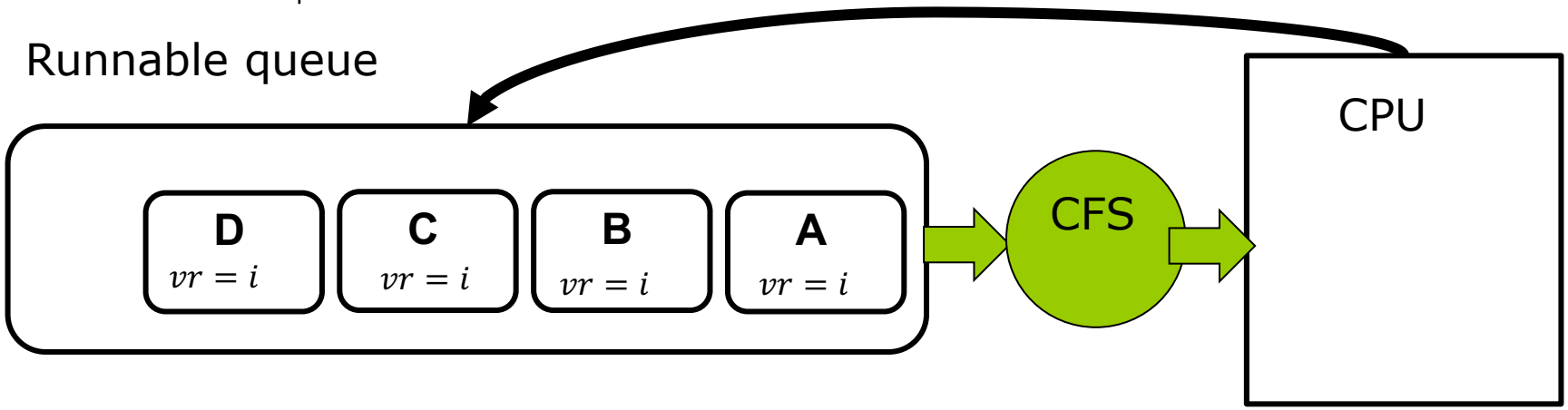
- CFS จะทำ Context Switching โดย copy state ของ Task j จาก PCB เข้าสู่ CPU แล้วให้ CPU ประมวลผล Task j ต่อไป

ตัวอย่าง CFS Scheduling

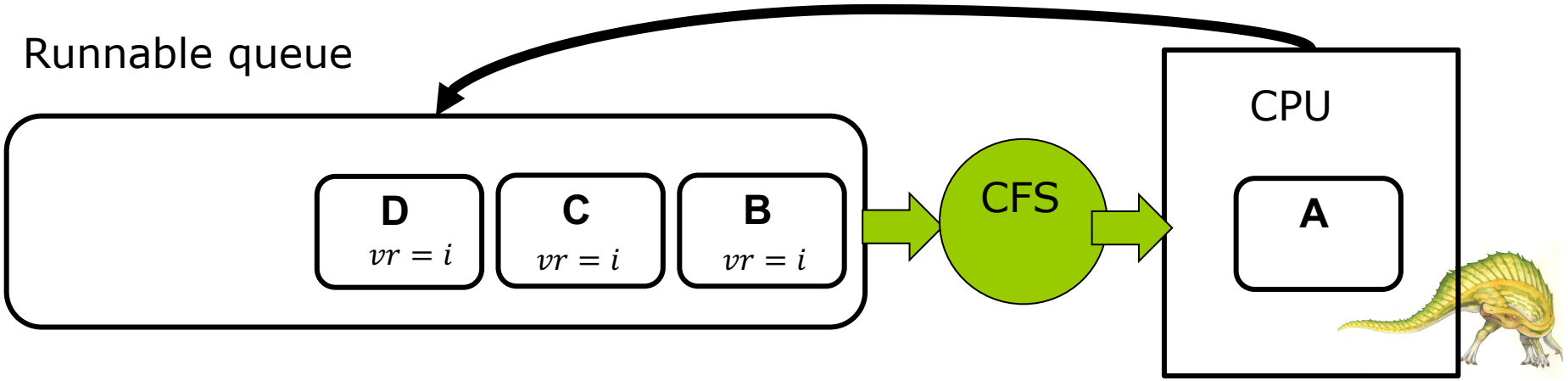
- เราจะรันตัวอย่างที่ใช้อธิบาย Ideal CFS ใหม่
- เพื่อความง่าย สมมุติว่าไม่มีการใช้ I/O ในระหว่างช่วง time slice
  ดังนั้น ค่า t ก็จะเท่ากับค่า time_slice
- สมมุติว่า TL/N > MG
- กำหนดให้ค่า vruntime เริ่มต้นคือ $vr = i$

จำลอง CFS Scheduling ตย 1 สมมุติ TL=4

- TL = Targeted Latency = 4 ms และ N เริ่มต้น = 4
- เริ่มต้น (สมมุติว่าค่าเริ่ม ของ $vrt = i$)

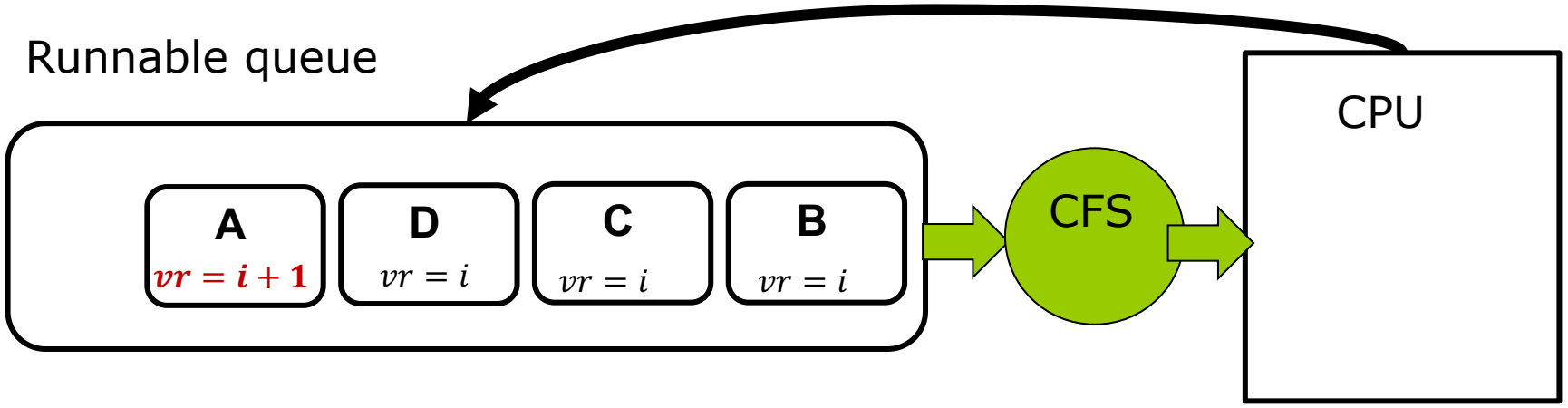


- เลือก A คำนวณค่า time_slice ของ A คือ 4/4 = 1 ms เมื่อ A จบ vr +=1 = i+1

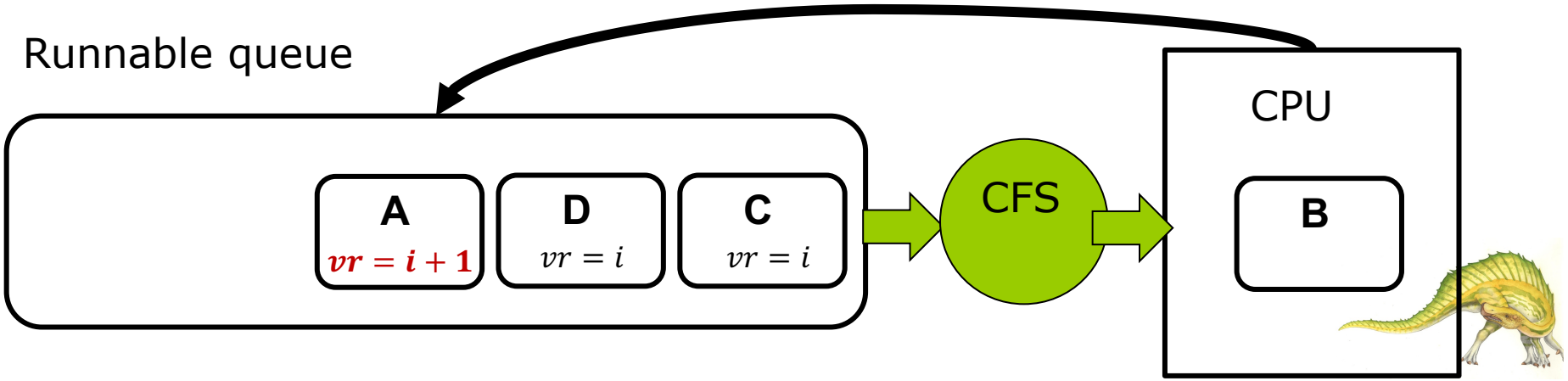


จำลอง Ideal CFS Scheduling

- A หมด time slice ค่า vruntime = i+1 ms เอากลับมาที่ run queue

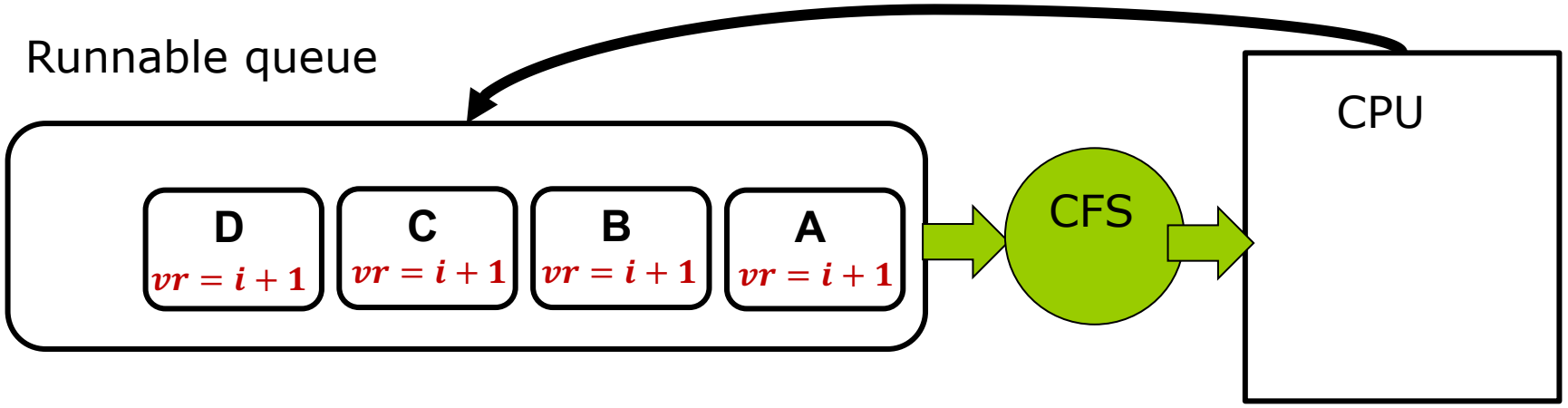


- เลือก B คำนวณค่า time_slice ของ B คือ 4/4 = 1 ms

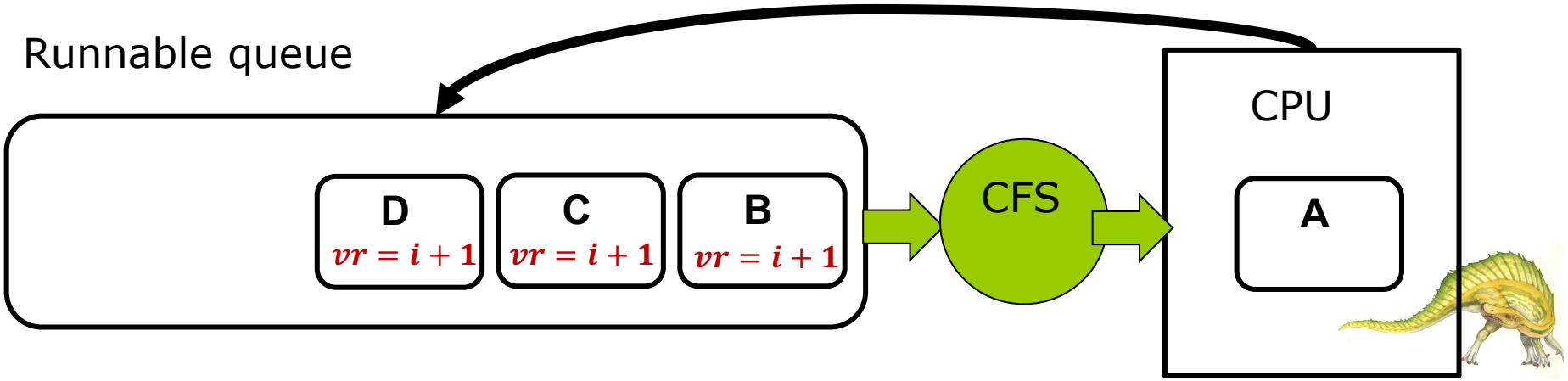


Ideal CFS Scheduling

- B หมด Time Slice CFS เลือก C และ D **เมื่อ D รันจบจะได้**

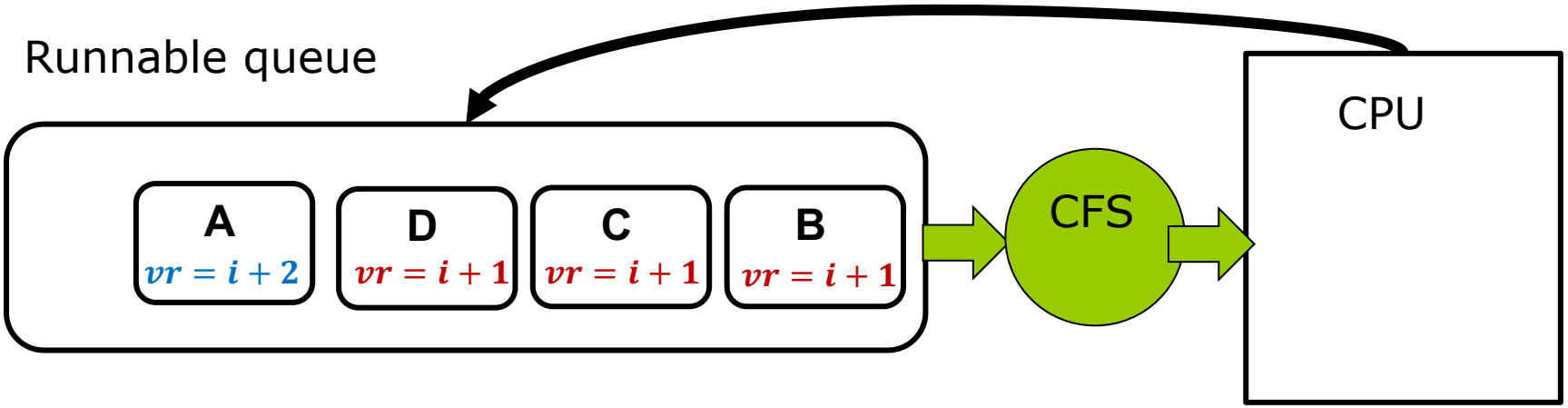


- เลือก A คำนวณค่า time_slice ของ A คือ 4/4 = 1 ms



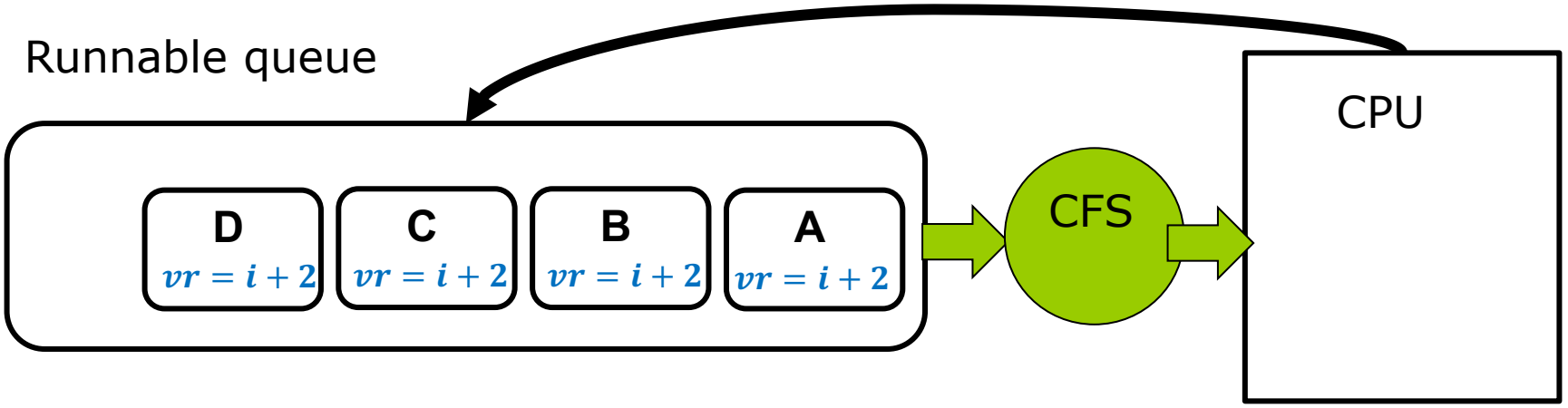
Ideal CFS Scheduling

- เมื่อ A จบ Time Slice จะได้ $vr = i + 2$

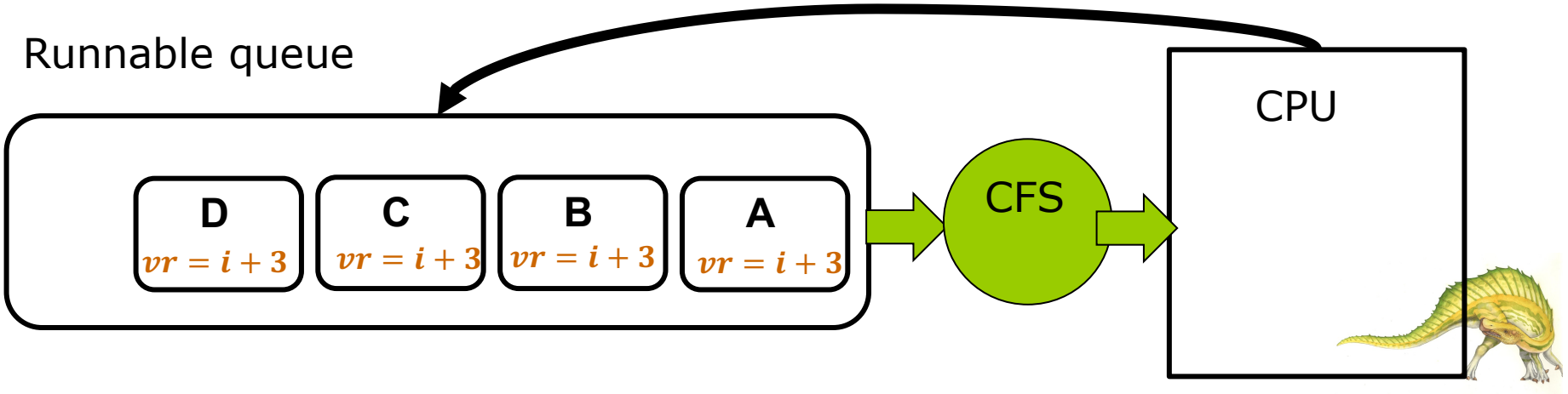


Ideal CFS Scheduling

- CPU รัน B C D เมื่อ D หมด Time Slice จะได้

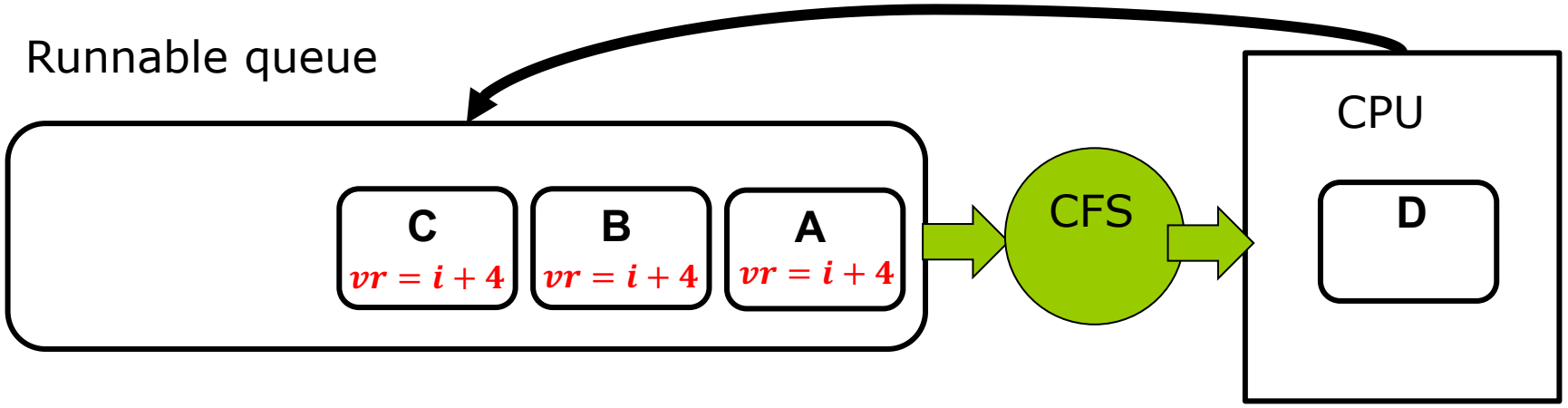


- CPU รัน A ถึง D ด้วยค่า time slice = 4/4 = 1 ms หลังจาก D หมด Time Sclice จะได้

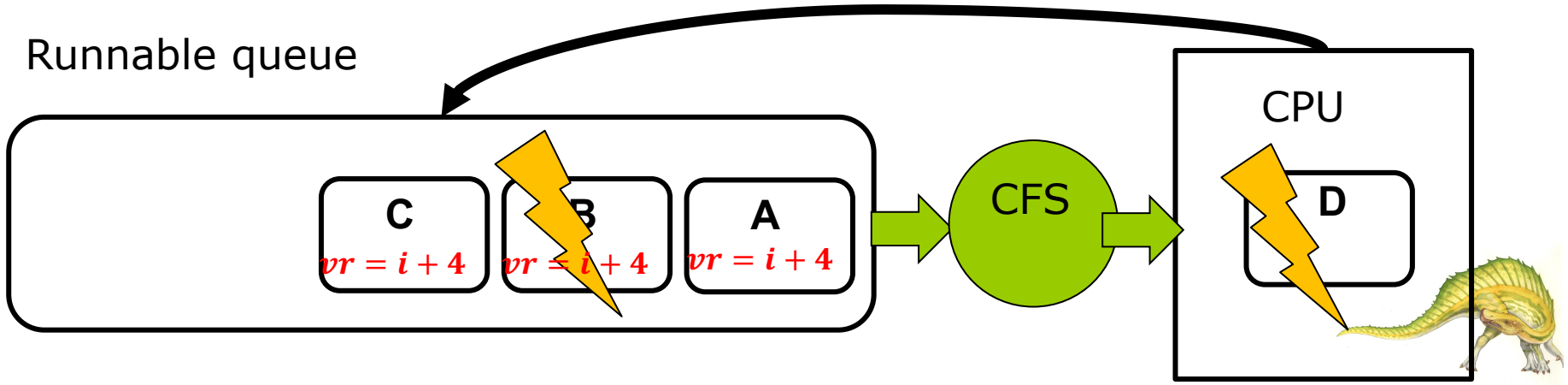


Ideal CFS Scheduling

- ถัดไปเมื่อ CPU รัน A B C แล้ว CFS เลือก D

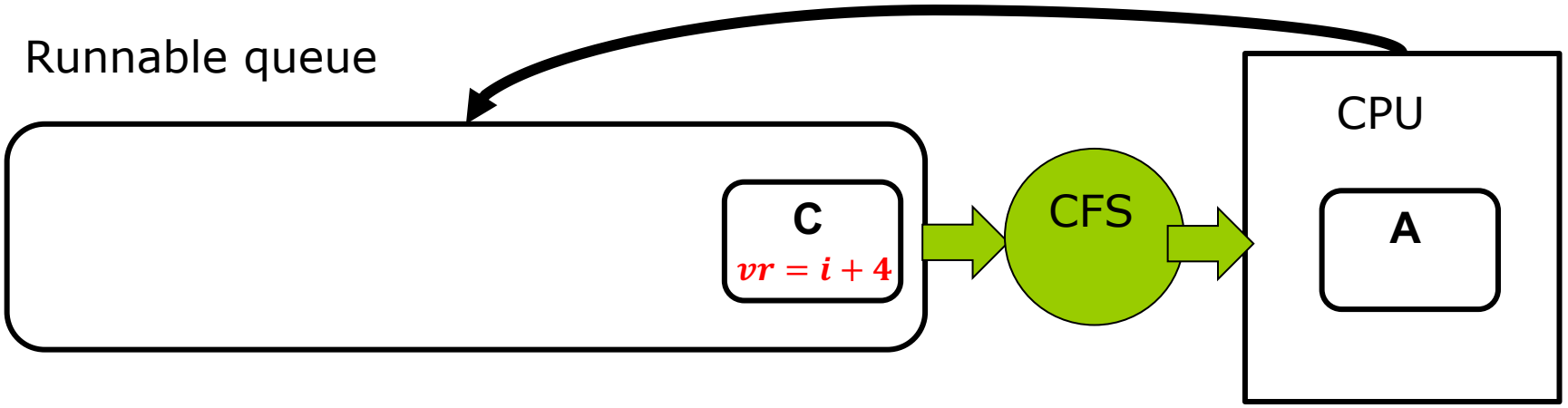


- สมมุติว่า B และ D เป็น Thread ใน Process เดียวกัน เมื่อ D exit → B D จึงจบ

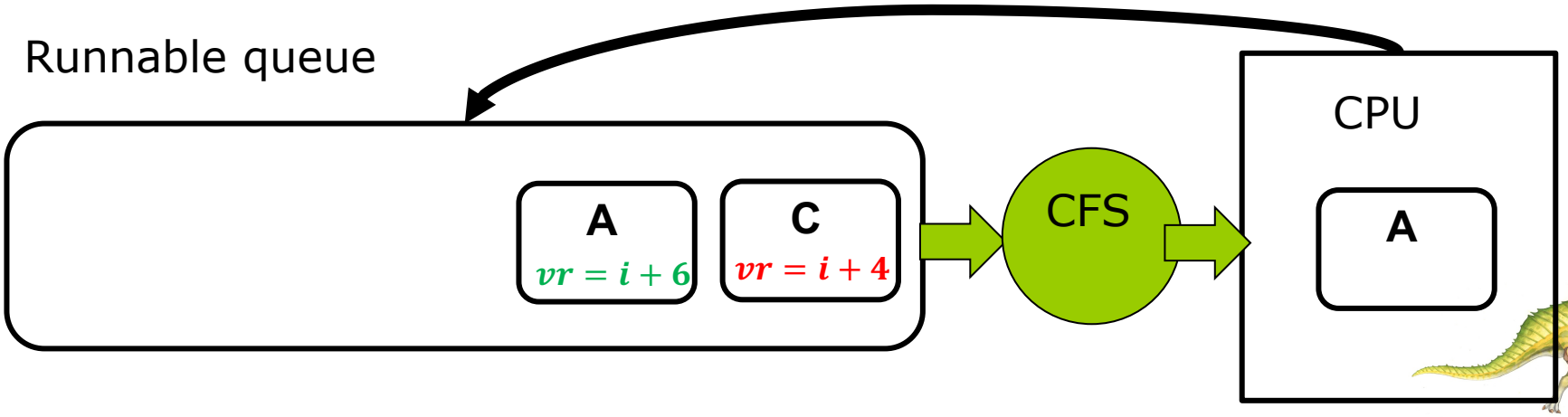


Ideal CFS Scheduling

- **ต่อจากนั้น** N=2, CFS เลือก A และ time_slice ของ A คือ 4/2 = 2 ms

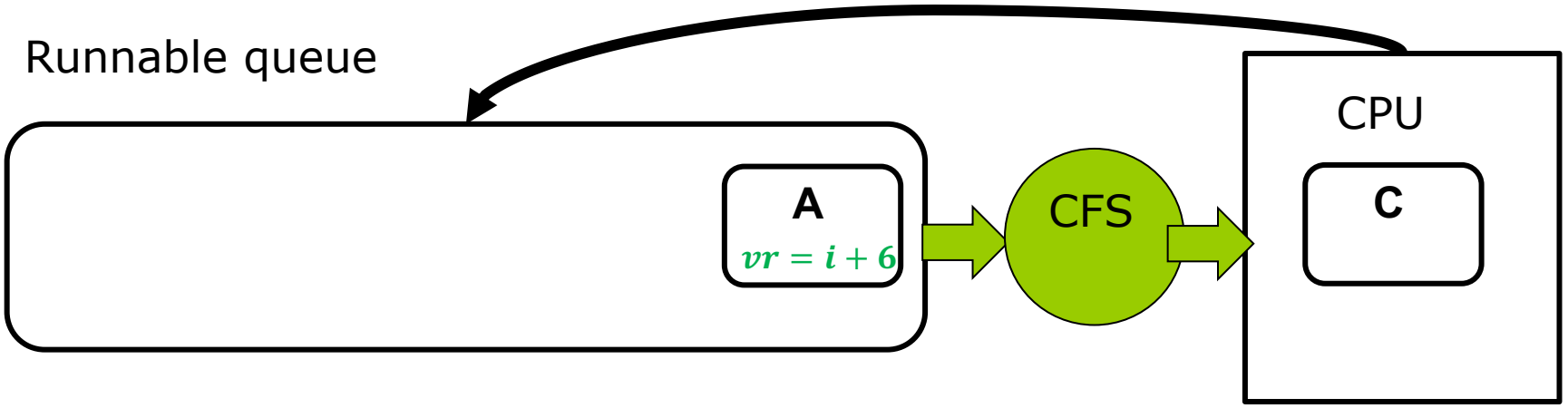


- เมื่อ A จบ Time Slice CFS update $vr = i + 6$

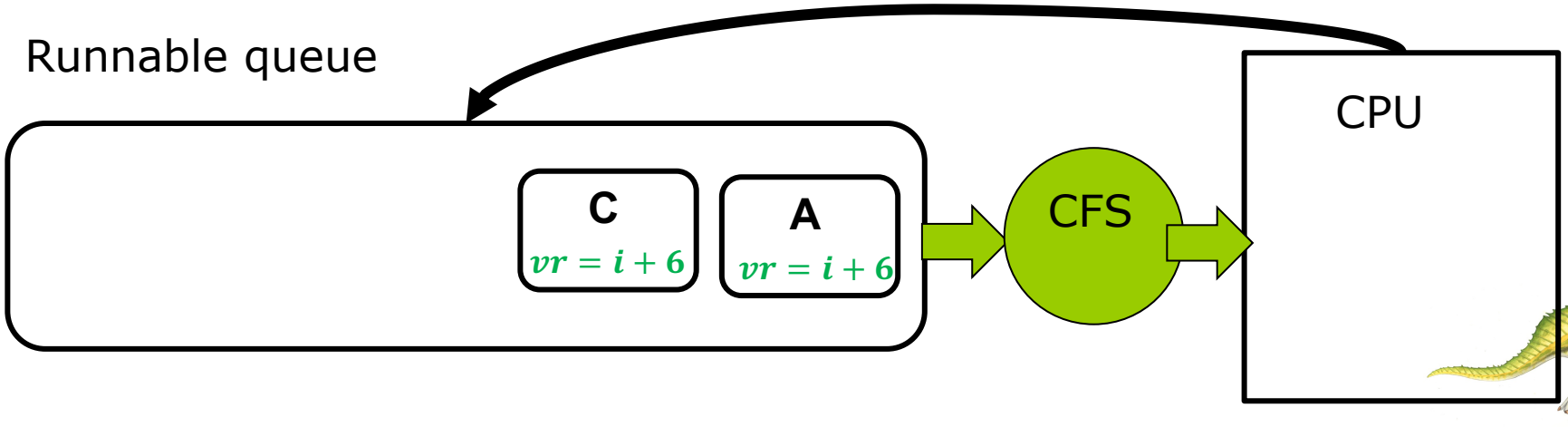


Ideal CFS Scheduling

- **ต่อจากนั้น** N=2, CFS เลือก C และ time_slice ของ C คือ 4/2 = 2 ms

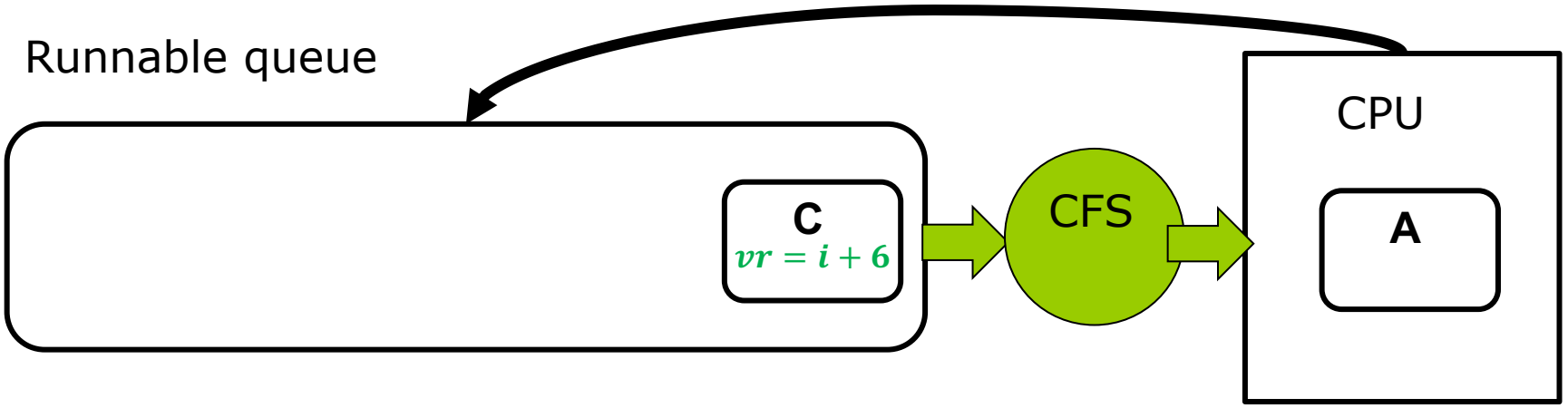


- และเมื่อ C รันจบ Time Slice **CFS update** $vr = i + 6$

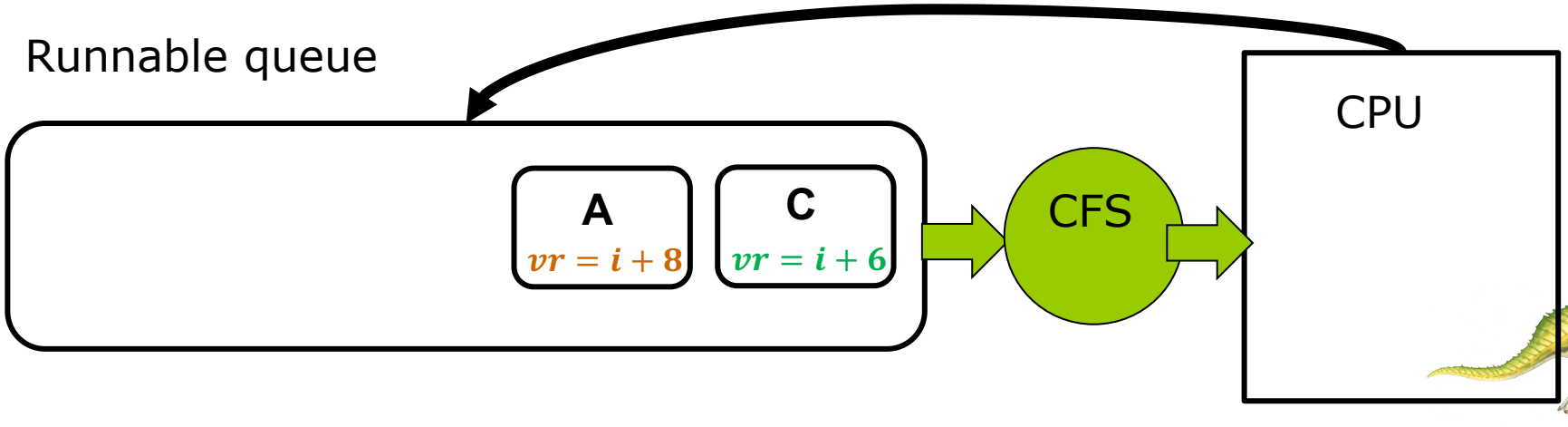


Ideal CFS Scheduling

- ที่ N=2, CFS เลือก A และ time_slice ของ A คือ 4/2 = 2 ms

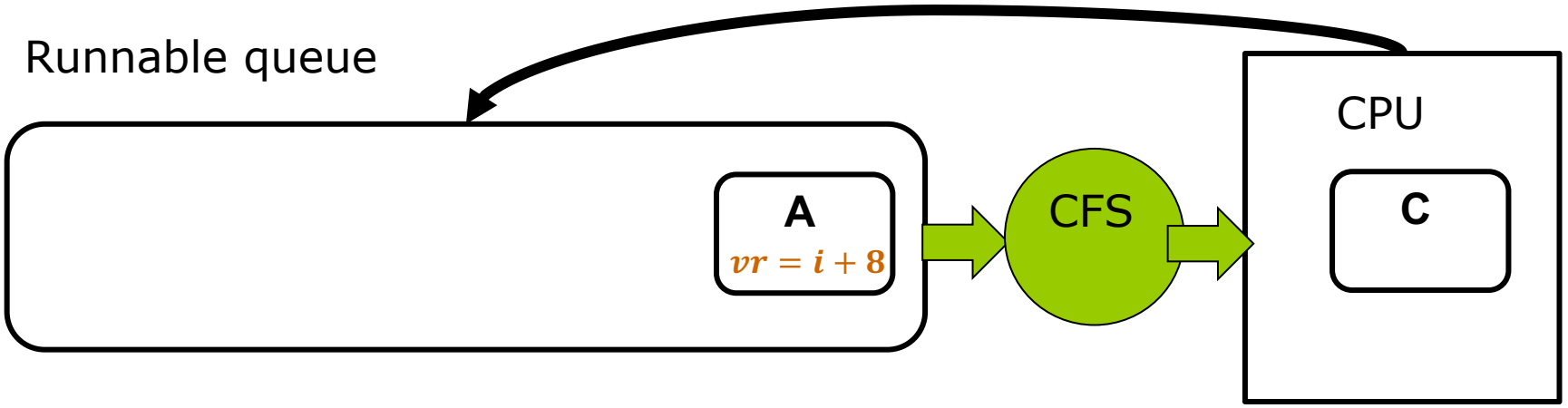


- เมื่อ A หมด Time Slice แล้ว CFS update $vr = i + 8$

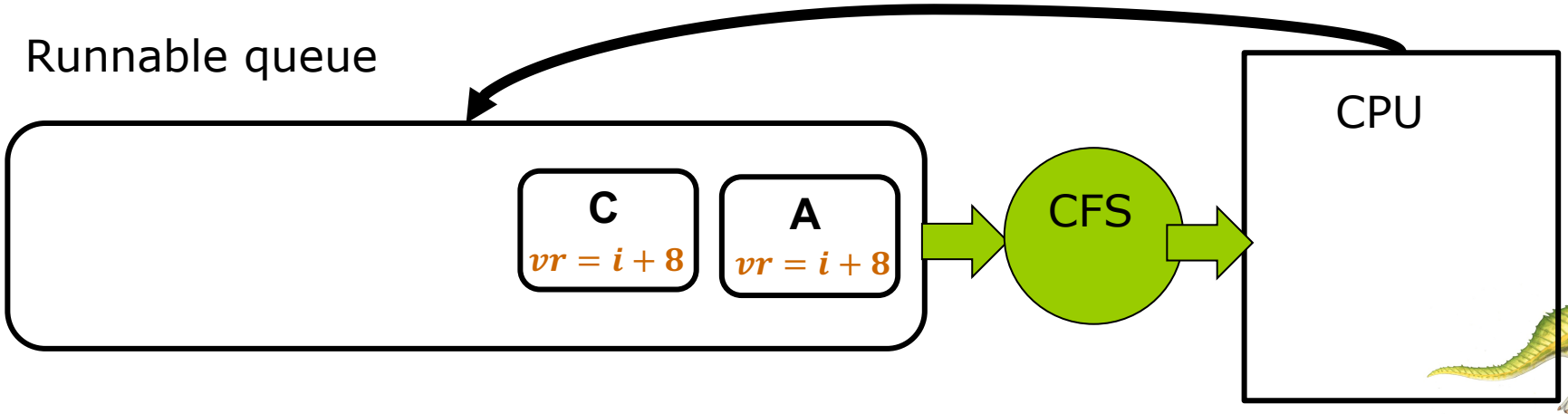


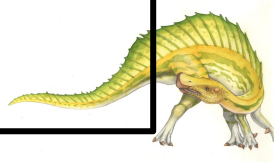
Ideal CFS Scheduling

- ที่ N=2, CFS เลือก C และ time_slice ของ C คือ 4/2 = 2 ms



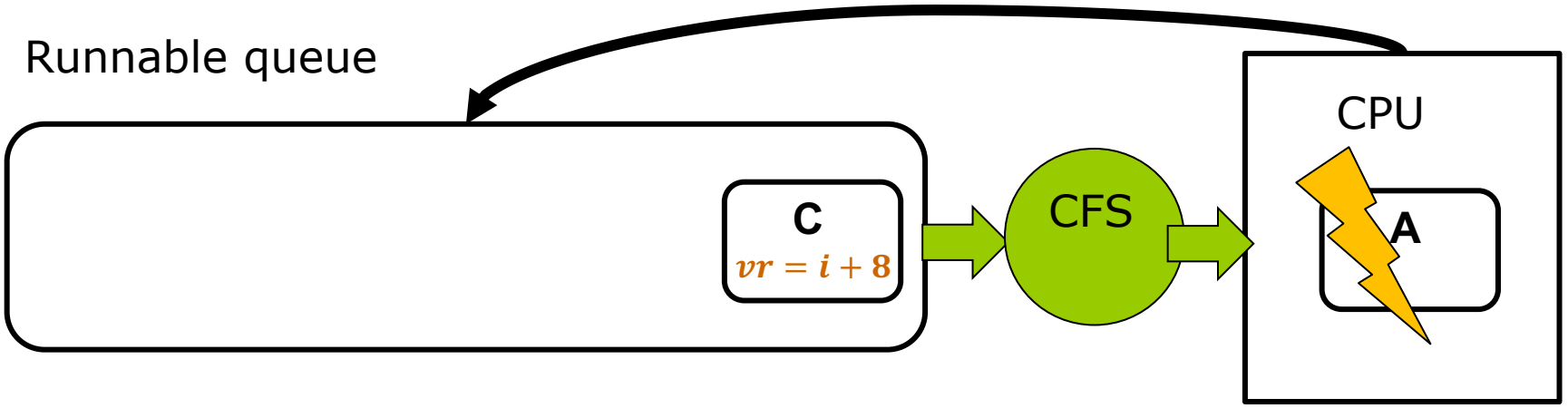
- เมื่อ C หมด Time Slice CFS ย้ายมันกลับมาที่ run queue และให้ $vr = i + 8$



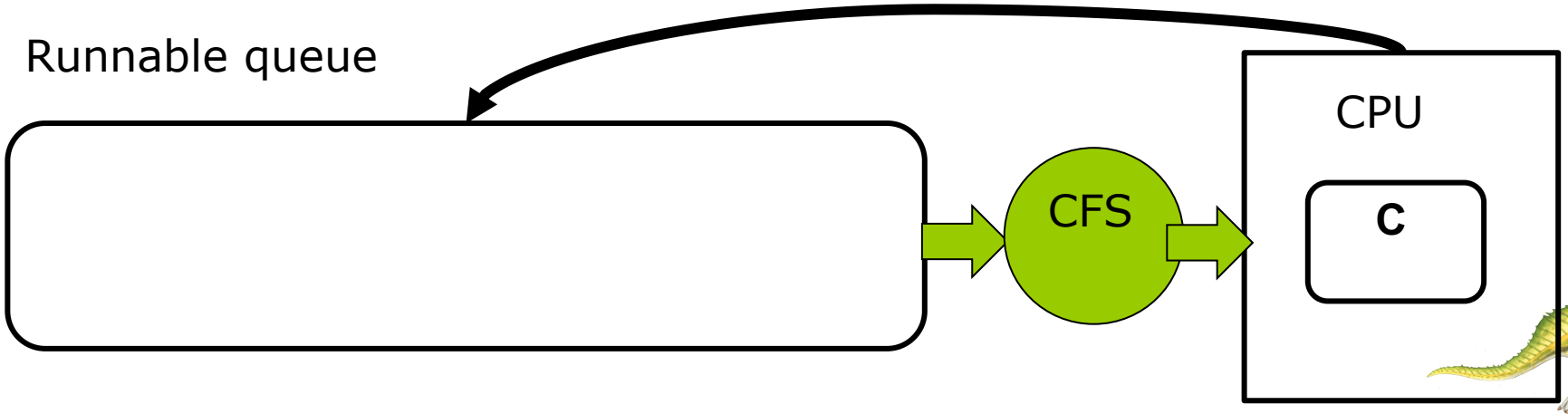


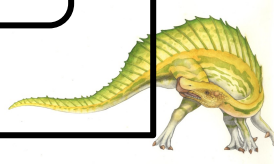
Ideal CFS Scheduling

- ที่ N=2, CFS เลือก A และ time_slice ของ A คือ 4/2 = 2 ms, แล้ว A รัน exit sys call



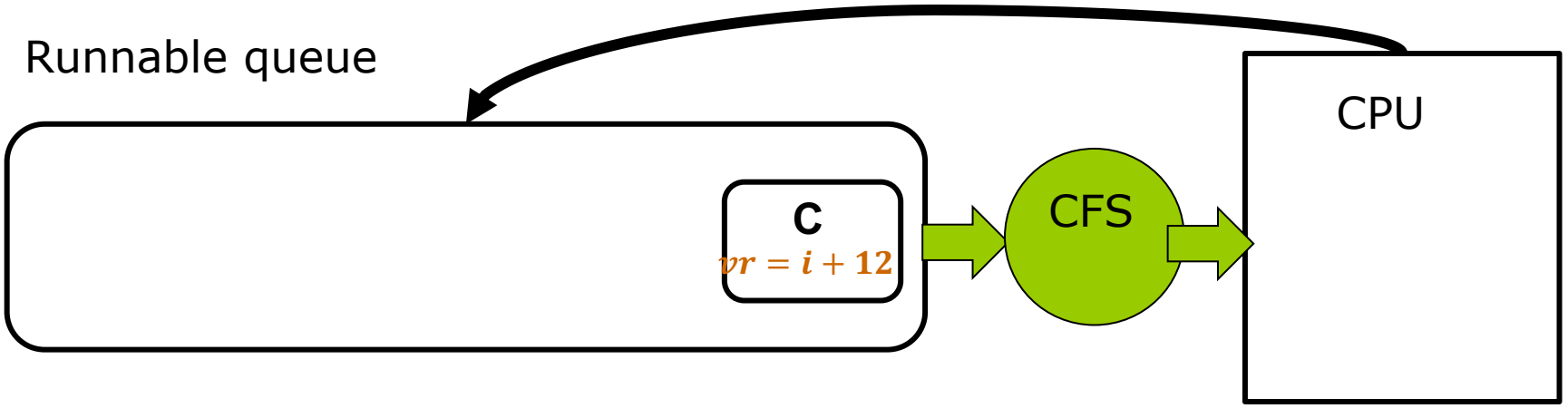
- **ต่อจากนั้น** N=1, CFS เลือก C และ time_slice ของ C คือ 4/1 = 4 ms



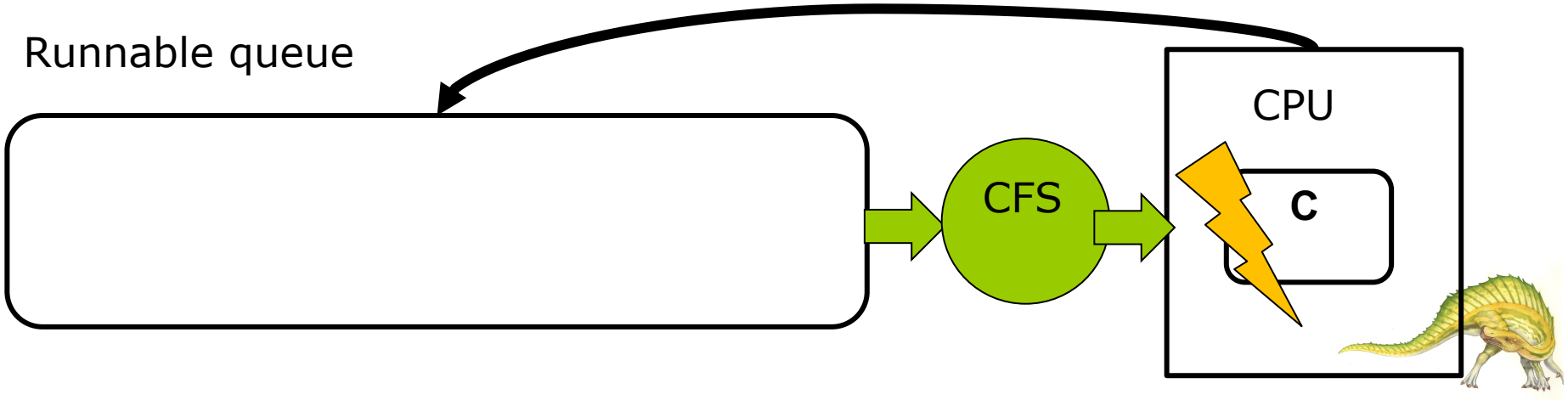


Ideal CFS Scheduling

- C รันหมด time slice และถูกนำกลับมา run queue CFS update $vr = i + 12$



- ถัดไป N=1, OS เลือก C และ time_slice ของ C คือ 4/1 = 4 ms, C รัน exit เมื่อจบ



หลักการ Completely Fair Scheduler 2

3. Task ทีเกิดใหม่จะมีเวลา CPU สะสมเท่ากับของ Task ที่มีเวลา CPU สะสมน้อยที่สุดใน run queue (ของ cpu core ที่มันถูกส่งไปรัน)

4. ในกรณี Task ถูกนำไปไว้ที่ wait queue เพื่อรอ I/O หรือ sleep

4.1 Task ที่รันคำสั่งขอใช้ I/O ถูกย้ายจาก CPU ไปอยู่ที่ wait queue เมื่อ I/O controller ทำงานเสร็จมันจะถูกย้ายกลับมาที่ run queue และเวลา CPU สะสมจะถูกกำหนดให้เท่ากับของ Task ใน run queue ที่มีเวลาสะสมน้อยที่สุด

4.2 Task ที่ถูกหยุดการทำงานชั่วคราว (sleep) และย้ายออกจาก run queue ไปอยู่ใน wait queue เมื่อตื่นแล้ว จะถูกนำกลับมารอ CPU ใน run queue อีกทีต้องกลับมารอที่ คิวอันดับเดิมเท่ากับตอนก่อนออกไป

(ใช้เวลา CPU สะสม เป็นเครื่องมือ)

-- เหมือนรอคิวอยู่แล้วต้องออกไปธุระนาน ผู้คุมคิวก็จดไว้ว่าตอนนี้อยู่อันดับอะไร เมื่อกลับเข้ามาคิวอาจเปลี่ยนไปแล้วแต่อนุญาตให้เข้ามาอยู่อันดับเดิมได้

2. ขั้นตอน CFS (Fork New Task)

- สมมุติว่า Task ***i*** กำลังประมวลผลอยู่ใน CPU แล้วเรียกใช้ fork() system call CPU จะประมวลผล fork system call routine เพื่อสร้าง Task ลูก สมมุติว่าชื่อ Task ***j***
- fork() จะเรียก CFS เพื่อพิจารณาว่าจะรัน Task ***j*** ที่ CPU core ใด (โดยพิจารณา load balancing) ซึ่งผลจะมีสองแบบคือ
  - Case 1: รัน Task ***j*** บน CPU core เดียวกันกับ Task ***i***
  - Case 2: รัน Task ***j*** บน CPU core อื่น

2. ขั้นตอน CFS (Fork New Task)

Case 1: รัน Task j บน CPU core x *เดียวกัน*กับ Task i

- CFS จะกำหนดให้ vruntime ของ Task j ($vruntime_j$) ให้เท่ากับค่า Minimum vruntime ใน run queue ของ CPU core x ($min\_vruntime_x$)

$$vruntime_j = min\_vruntime_x$$

- CFS นำ Task j ไปใส่ใน run queue ของ CPU core x
- fork() system call ประมวลผลต่อ (บน CPU core x) จนจบแล้วคืนการประมวลผลให้ Task i (Parent Task) ประมวลผลต่อ

2. ขั้นตอน CFS (Fork New Task)

Case 2: รัน Task j บน CPU core y อื่น

- CFS จะกำหนดให้ vruntime ของ Task j ($vruntime_j$) ให้เท่ากับค่า Minimum vruntime ใน run queue ของ CPU core y ($min\_vruntime_y$)

$$vruntime_j = min\_vruntime_y$$

- CFS นำ Task j ไปใส่ใน run queue ของ CPU core y
- fork() system call ประมวลผลต่อ (บน CPU core x) จนจบแล้วคืนการประมวลผลให้ Task i (Parent Task) ประมวลผลต่อ

อ้างอิง

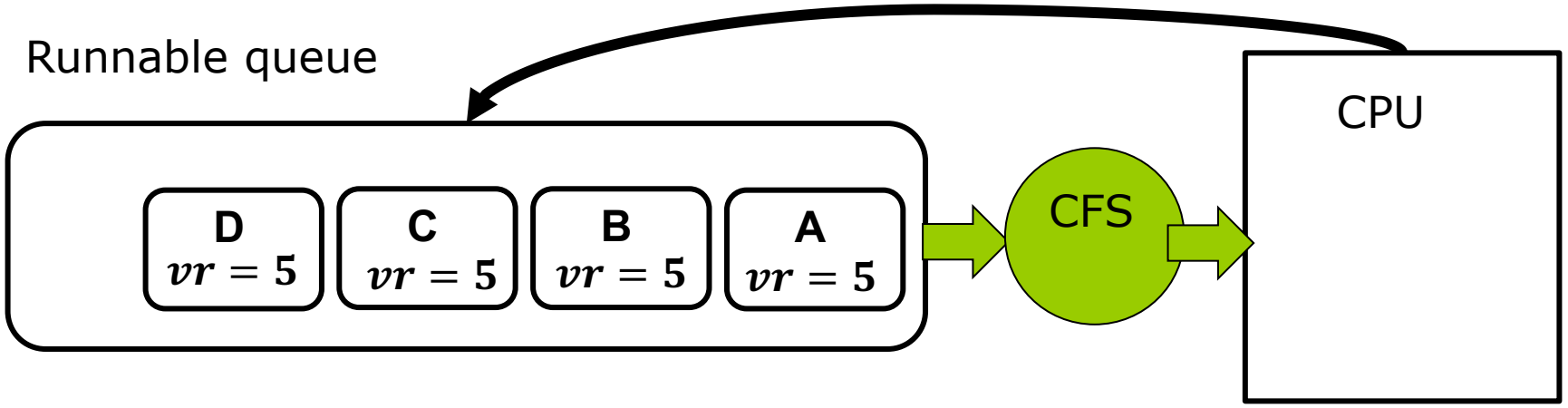
scheduling - How does the Completely Fair Scheduler prevent starvation if it's always taking the process with the lowest vruntime? - Unix & Linux Stack Exchange

จำลอง CPU Scheduling ตย 2 สมมุติ TL=20

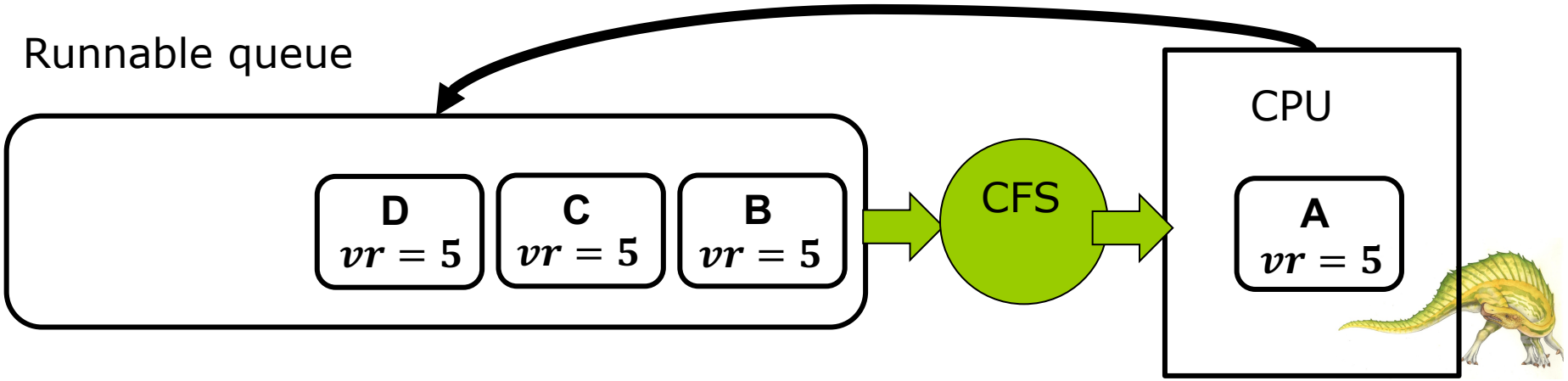
- เราจะแสดงตัวอย่างการจำลอง CFS CPU scheduling อีกครั้ง
- แต่คราวนี้เราจะเพิ่มการ สร้าง (fork) task ใหม่เข้าสู่ run queue
- ข้อกำหนด
  - จำลองโดยใช้**ขั้นตอน CFS 1 และ 2**
  - มีค่า TL = 20 ms
  - มีค่า MG = 4 ms
  - มี Tasks (อาจเป็น Process หรือ Thread) เริ่มต้น 4 Tasks
  - สมมุติว่าค่า vruntime (หรือ vr) เริ่มต้นของทุก Task คือ $vr = 5$

จำลอง CFS Scheduling สมมุติ TL=20

- เริ่มต้น N = 4

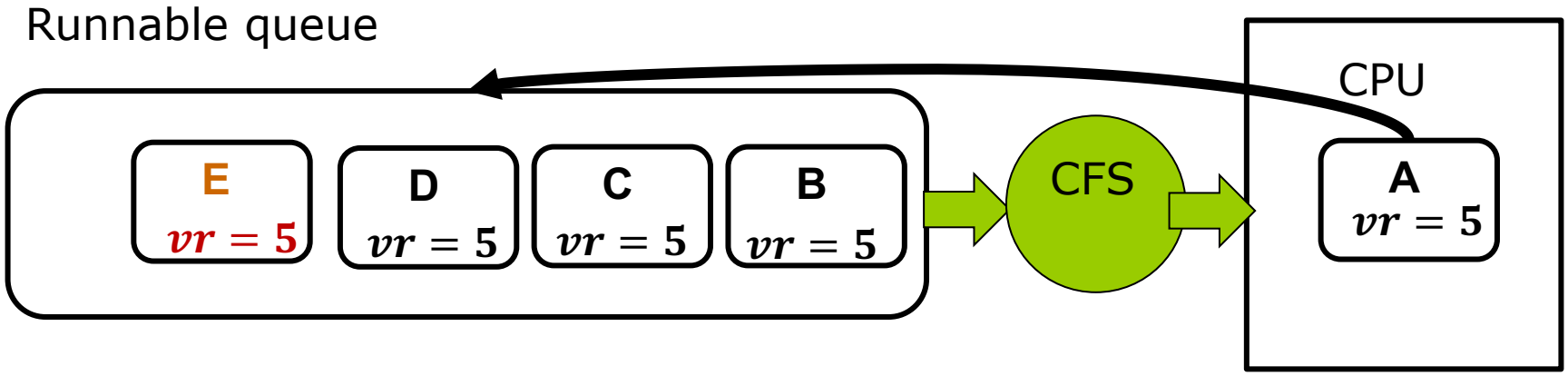


- เลือก A คำนวณค่า time_slice ของ A คือ $ts_A = \frac{20}{4} = 5$

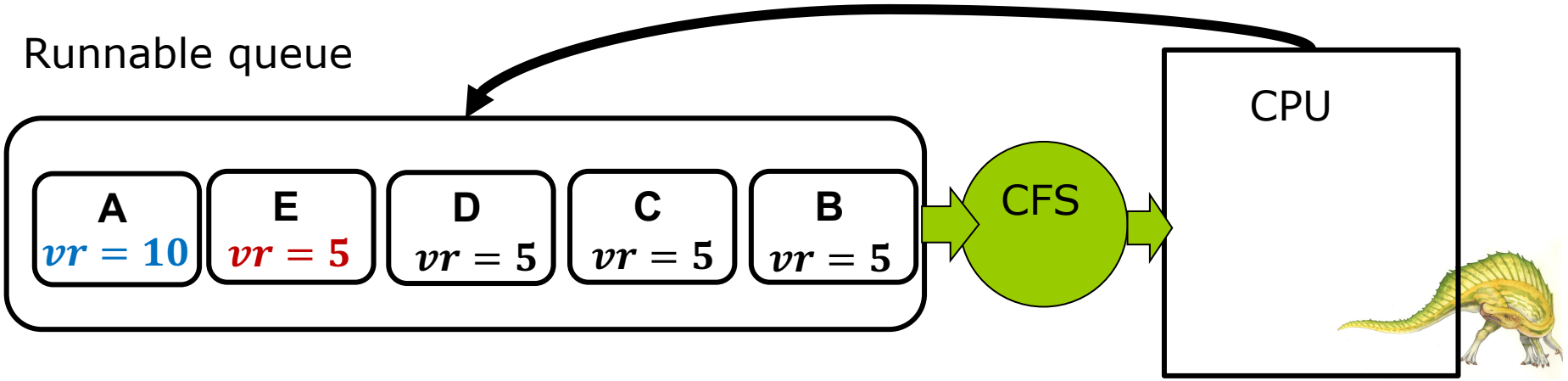


จำลอง CFS Scheduling สมมุติ TL=20

- สมมุติว่า A fork new Task “E” จะมี item ใหม่ใน $vr = Min(vr$ ของ $Tasks$ ใน $run\ queue) = 5$ (E มีโอกาสได้แข่งกับ task ต้นคิว)

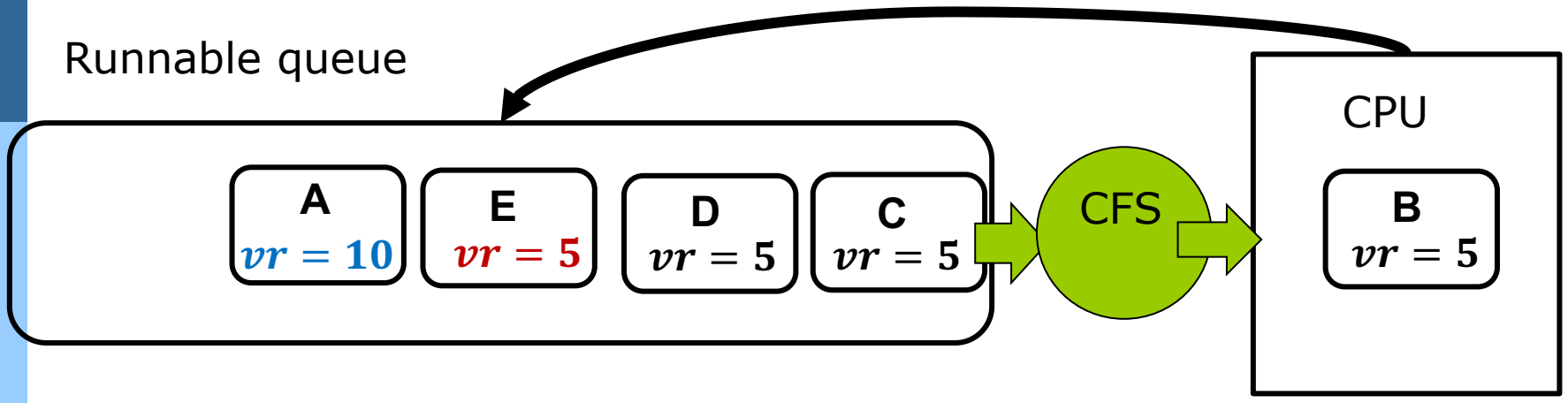


- เมื่อ A หมด time slice และ CFS เปลี่ยน vr ของ A เป็น $vr = 5 + 5 = 10$

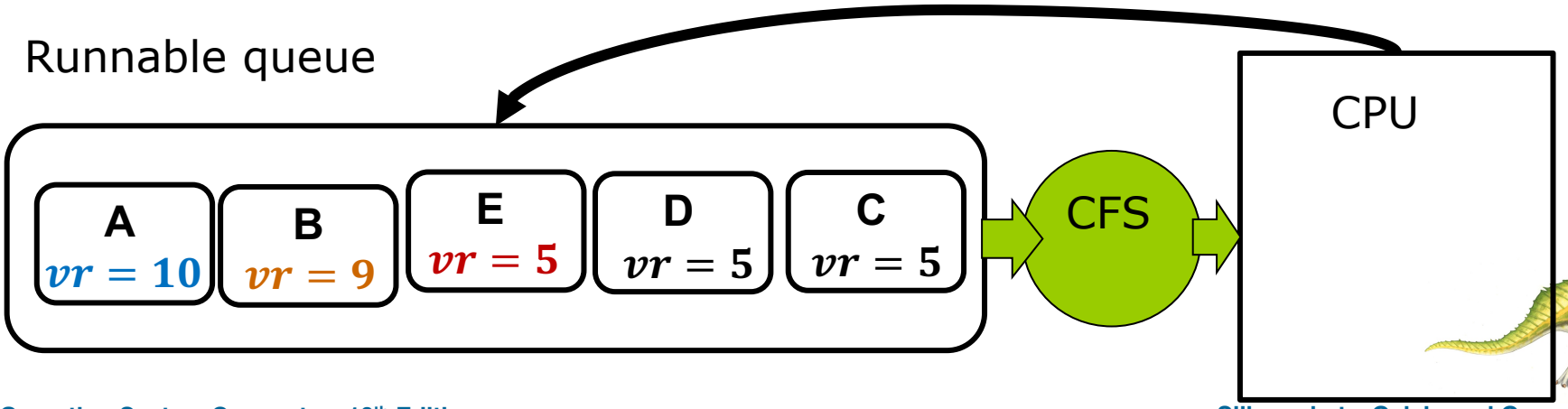


จำลอง Ideal CFS Scheduling สมมุติ TL=20

- CFS เลือก “B” และ N=5 ดังนั้น time_slice ของ B คือ $ts_B = \frac{20}{5} = 4\ ms$



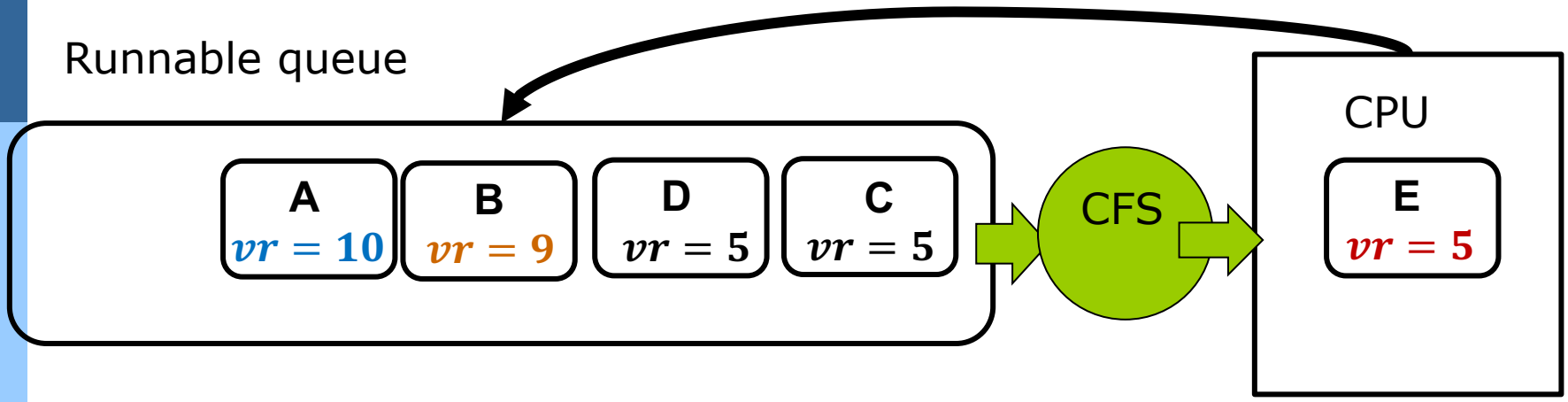
- เมื่อ B หมด time slice และ CFS เปลี่ยน vr ของ B เป็น $vr = 5 + 4 = 9$



จำลอง Ideal CFS Scheduling สมมุติ TL=20

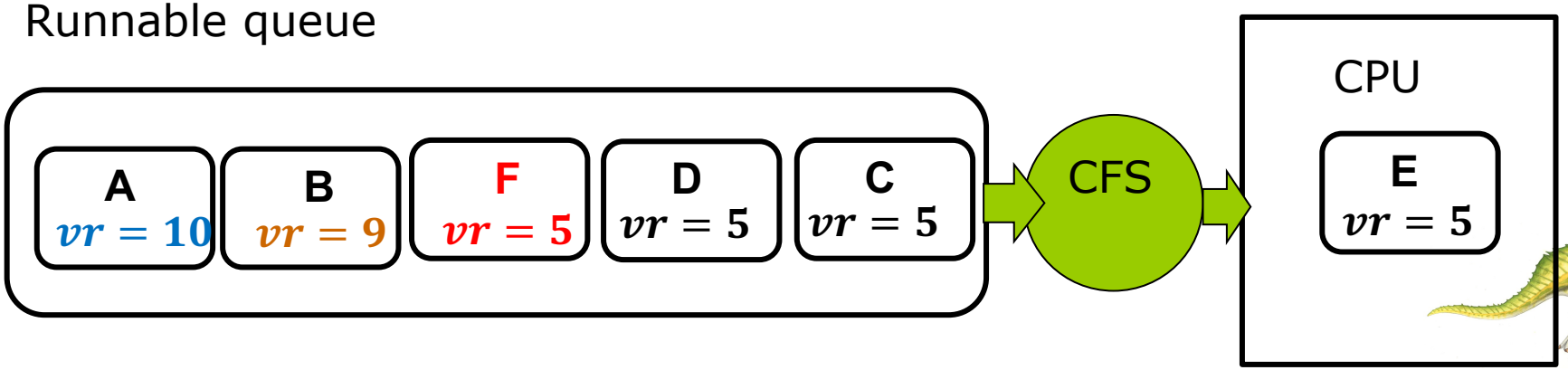
- CFS เลือก “E” และ N=5 ดังนั้น time_slice ของ E คือ $ts_E = \frac{20}{5} = 4\,ms$

Runnable queue

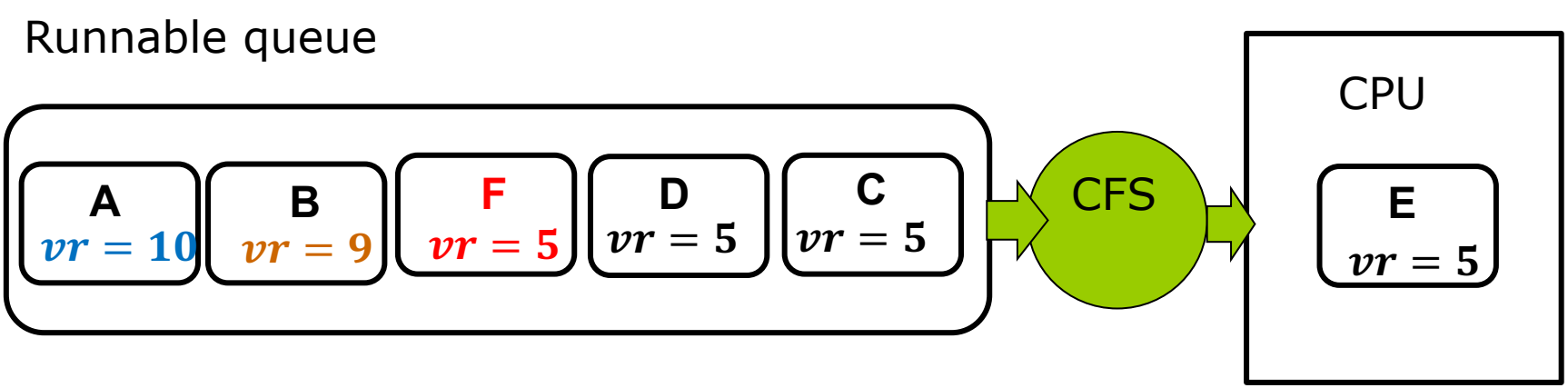


- สมมุติว่า CFS assign Task ใหม่ “F” (parent อยู่ที่ CPU core อื่น) มาที่ run queue $\boldsymbol{vr = Min(vr}$ ของ $\boldsymbol{Tasks}$ ใน $\boldsymbol{run\ queue) = 5}$

Runnable queue



จำลอง Ideal CFS Scheduling สมมุติ TL=20



1. จากภาพนี้ ถามว่าเมื่อ E จบ Time slice (=4) ค่า vruntime ของ E = ?
2. เมื่อ E หมด Time Slice จงเขียนภาพ รายการ Tasks ใน run queue ใหม่
3. สมมุติว่า CFS เลือก C เข้าประมวลผล ค่า Time Slice ของ C จะเป็นเท่าไร ? (Trick Question)
4. หลังจาก E หมด Time Slice ค่าเวลารวมในการประมวลผลทุก Task ในรอบถัดไป (หมายถึง CFS เลือก C D F A B E) จะเป็นเท่าไร

หลักการ Completely Fair Scheduler 2

3. Task ที่เกิดใหม่จะมีเวลา CPU สะสมเท่ากับของ Task ที่มีเวลา CPU สะสมน้อยที่สุดใน run queue (ของ cpu core ที่มันถูกส่งไปรัน)

4. ในกรณี Task ถูกนำไปไว้ที่ wait queue เพื่อรอ I/O หรือ sleep

4.1 Task ที่รันคำสั่งขอใช้ I/O ถูกย้ายจาก CPU ไปอยู่ที่ wait queue เมื่อ I/O controller ทำงานเสร็จมันจะถูกย้ายกลับมาที่ run queue และเวลา CPU สะสมจะถูกกำหนดให้เท่ากับของ Task ใน run queue ที่มีเวลาสะสมน้อยที่สุด

4.2 Task ที่ถูกหยุดการทำงานชั่วคราว (sleep) และย้ายออกจาก run queue ไปอยู่ใน wait queue เมื่อตื่นแล้ว จะถูกนำกลับมารอ CPU ใน run queue อีกทีต้องกลับมารอที่ <u>คิวอันดับเดิมเท่ากับตอนก่อนออกไป</u>

(ใช้**เวลา CPU สะสม** เป็นเครื่องมือ)

-- เหมือนรอคิวอยู่แล้วต้องออกไปธุระนาน ผู้คุมคิวก็จดไว้ว่าตอนนี้อยู่อันดับอะไร เมื่อกลับเข้ามาคิวอาจเปลี่ยนไปแล้วแต่อนุญาตให้เข้ามาอยู่อันดับเดิมได้

3. ขั้นตอน CFS (wait for I/O)

Case 1: Task i บน CPU core x *รันคำสั่งที่ต้องรอ I/O หรือ Sleep*

- สมมุติว่า Task i กำลังประมวลผลอยู่ใน CPU แล้วเรียกใช้คำสั่งที่ต้องรอ I/O หรือ sleep CFS จะนำ Task i ไปใส่ใน wait queue เพื่อรอให้จบ I/O หรือ sleep activities
- เมื่อ activities ที่รอเสร็จแล้ว Linux จะนำ Task i กลับเข้ามาไว้ที่ run queue อีกครั้งหนึ่ง และกำหนดค่า vruntime เป็น

$$vruntime_i = min\_vruntime_x$$

- กำหนดให้ $vruntime_i$ หมายถึง vruntime ของ Task i
- และ $min\_vruntime_x$ หมายถึง Min vruntime ของ Tasks ใน run queue ของ CPU core x

scheduling - How does the Completely Fair Scheduler prevent starvation if it's always taking the process with the lowest vruntime? - Unix & Linux Stack Exchange

3. ขั้นตอน CFS (wait for I/O)

Case 2: Task j ใน run queue *ถูกสั่งให้รอ I/O หรือ Sleep*

- ผู้ใช้ หรือ Task ที่กำลังประมวลผล หรือ Kernel สามารถสั่งให้ Task j ที่กำลังรออยู่ใน run queue ของ CPU core x หยุดการประมวลผลชั่วคราว (sleep) และส่งมันไปรอใน wait queue
- ก่อนจะย้าย Task j CFS จะพิจารณาว่า Task j อยู่ในอันดับใดใน run queue โดยคำนวณค่า offset Task j

$$\mathbf{offset_j} = \mathbf{vruntime}_j - min\_vruntime_x$$

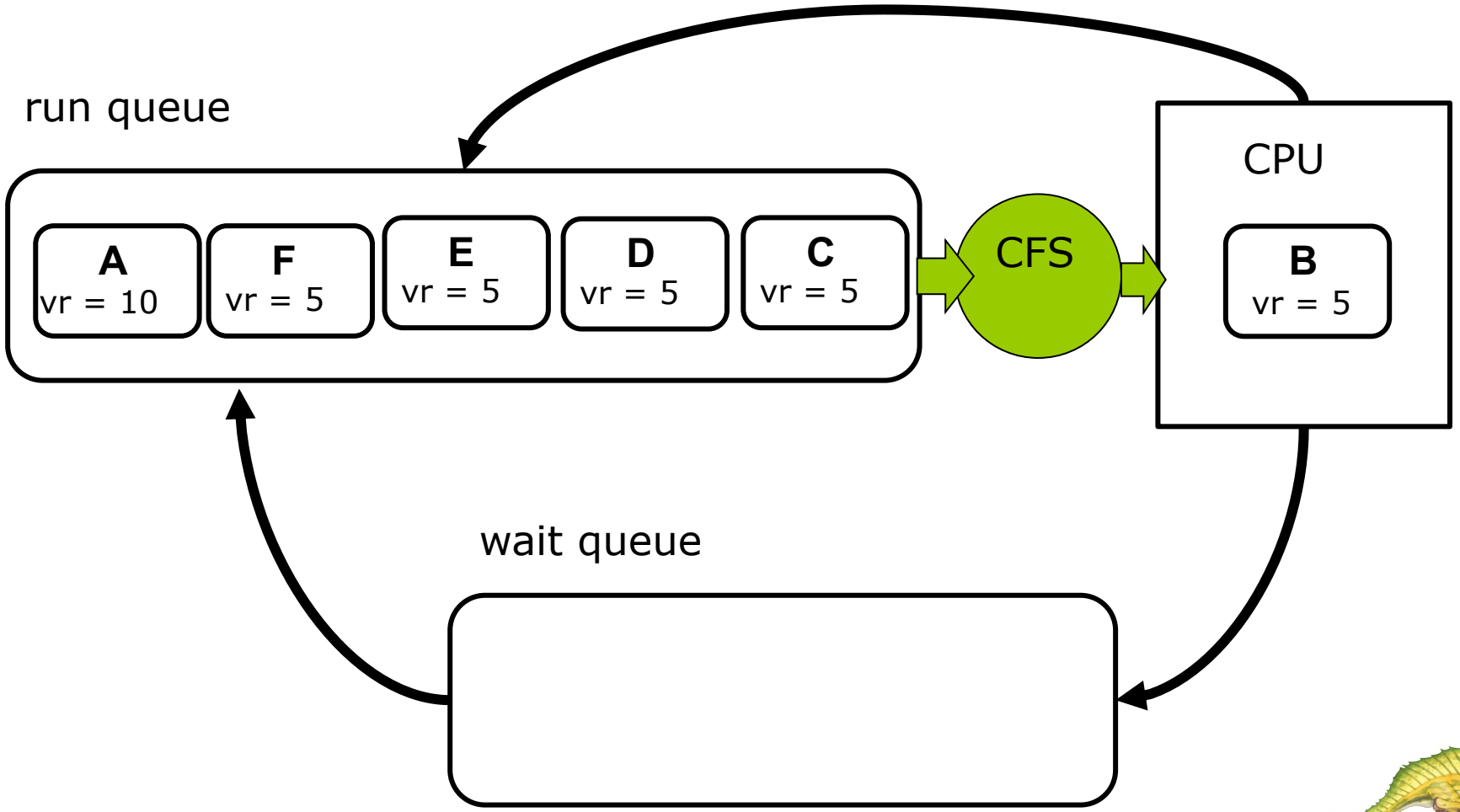
- เมื่อ sleep เสร็จแล้ว Linux จะนำ Task j กลับเข้ามาไว้ที่ run queue อีกครั้งหนึ่ง และกำหนดค่า vruntime เป็น

$$\mathbf{vruntime}_j = min\_vruntime_x + \mathbf{offset_j}$$

https://stackoverflow.com/questions/11297285/how-can-the-linux-cfs-scheduler-prevent-a-task-with-a-very-small-vruntime-from-s

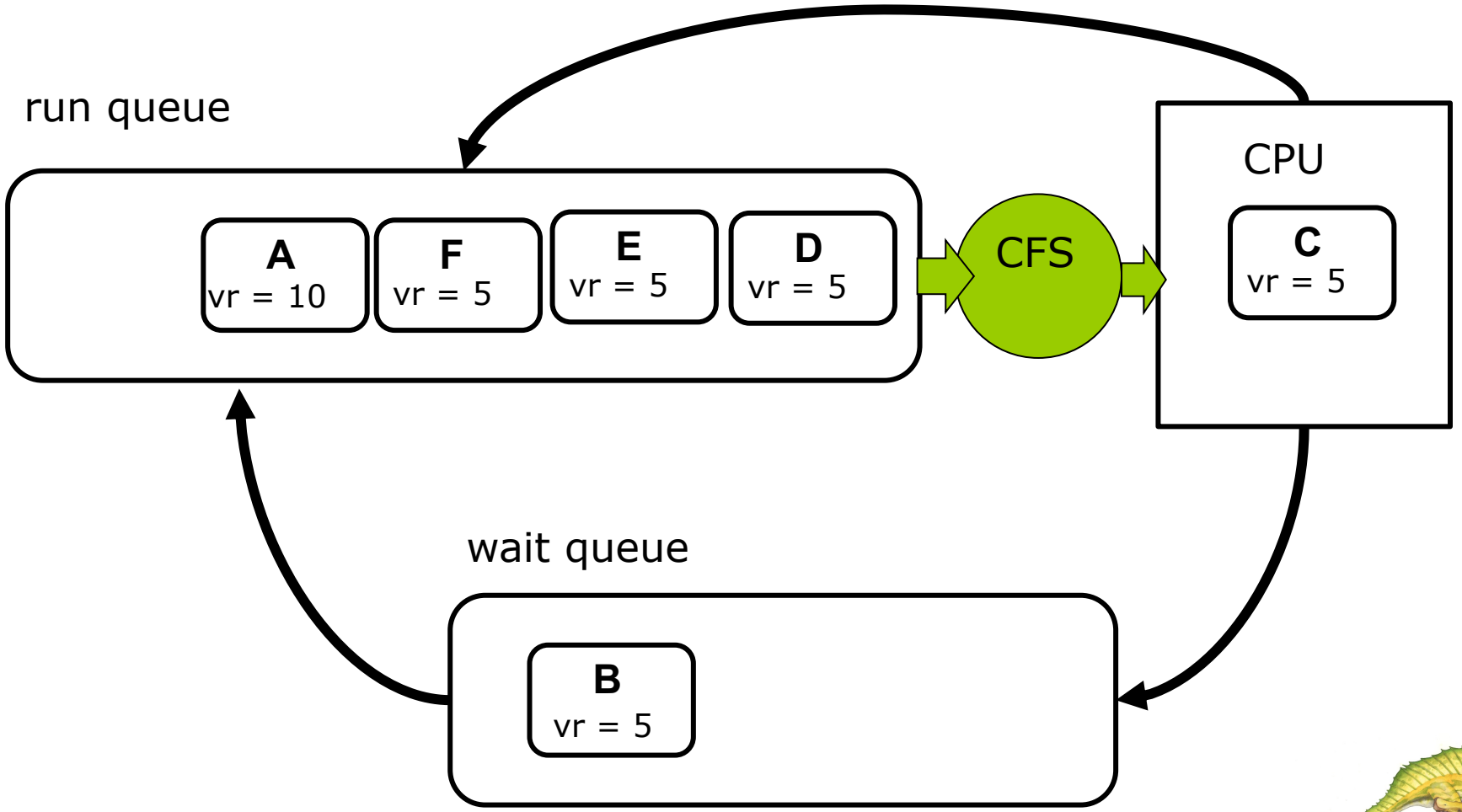
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- สมมุติว่า B (vr = 5) กำลังใช้ CPU และ รันคำสั่งเพื่ออ่านข้อมูลจาก I/O



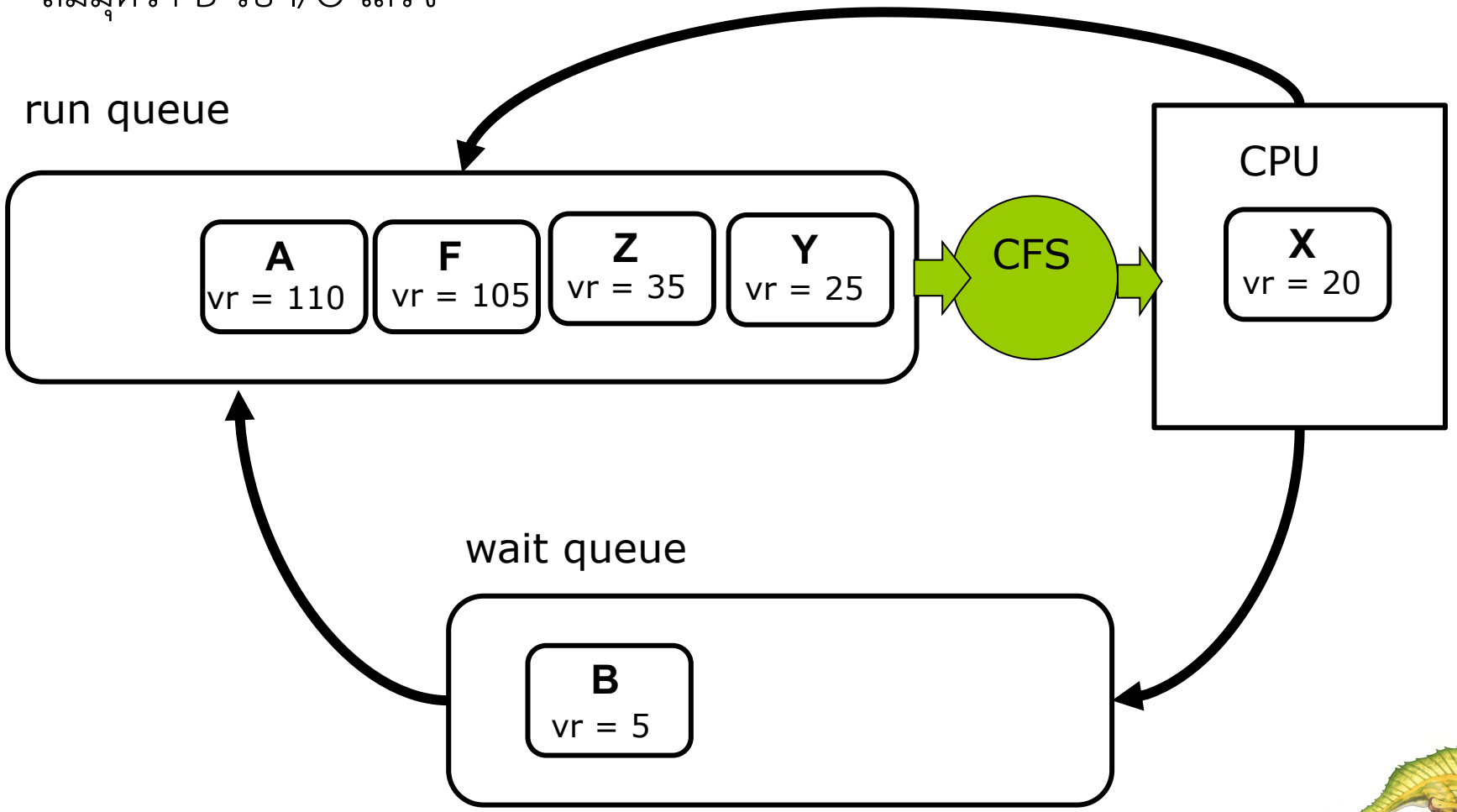
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- B (vr = 5) ถูกนำไปไว้ใน wait queue และ CFS เลือก C เข้าใช้ CPU



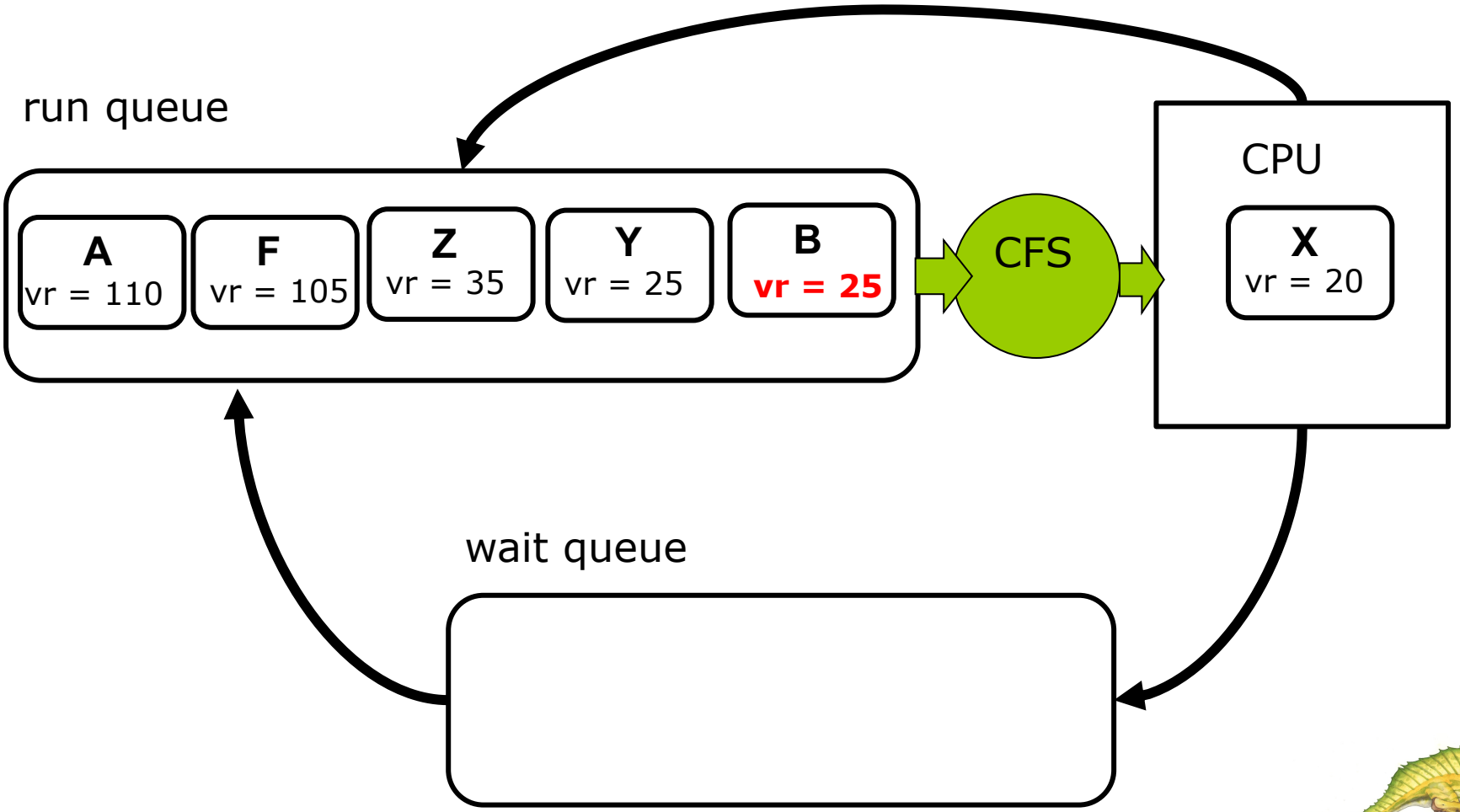
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- หลังจากนั้น 100 ms ผ่านไป run queue มีสภาพใหม่ดังนี้
- สมมุติว่า B รอ I/O เสร็จ



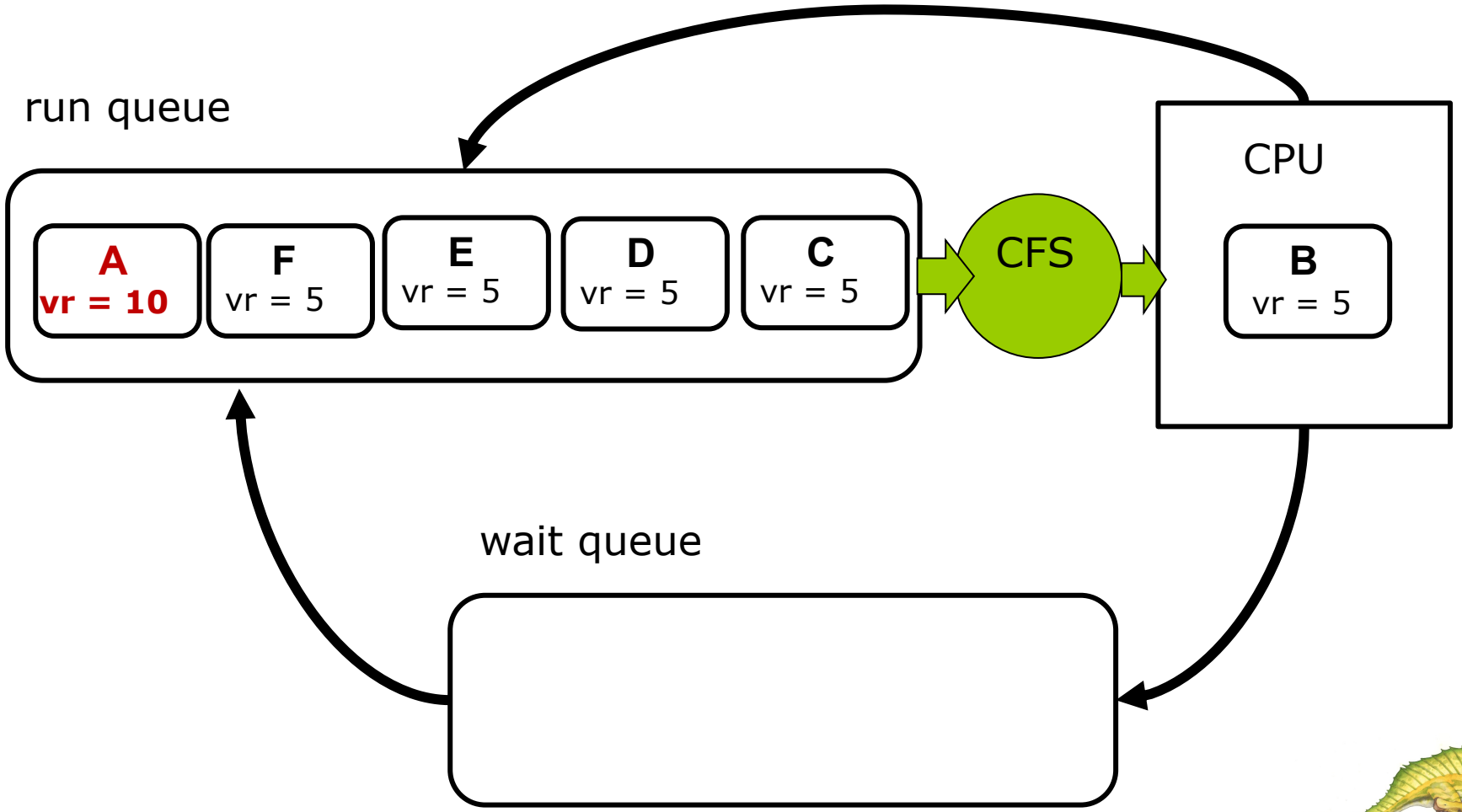
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- CFS จะกำหนดให้ B มี vruntime คือ Min vr ของ task ใน run queue = 25



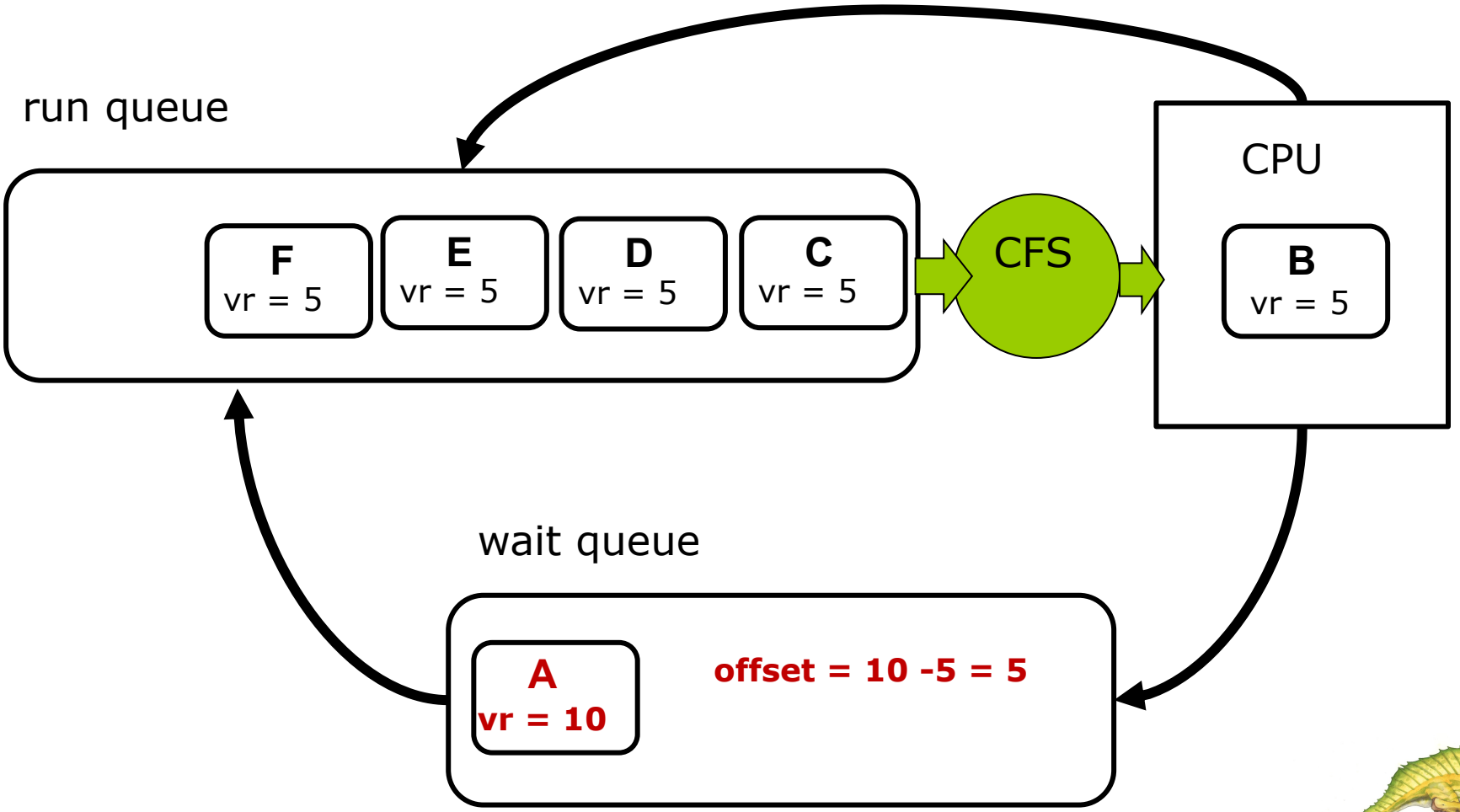
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- สมมุติว่า B (vr = 5) กำลังใช้ CPU และ รันคำสั่งเพื่อสั่งให้ A หยุดทำงานชั่วคราว



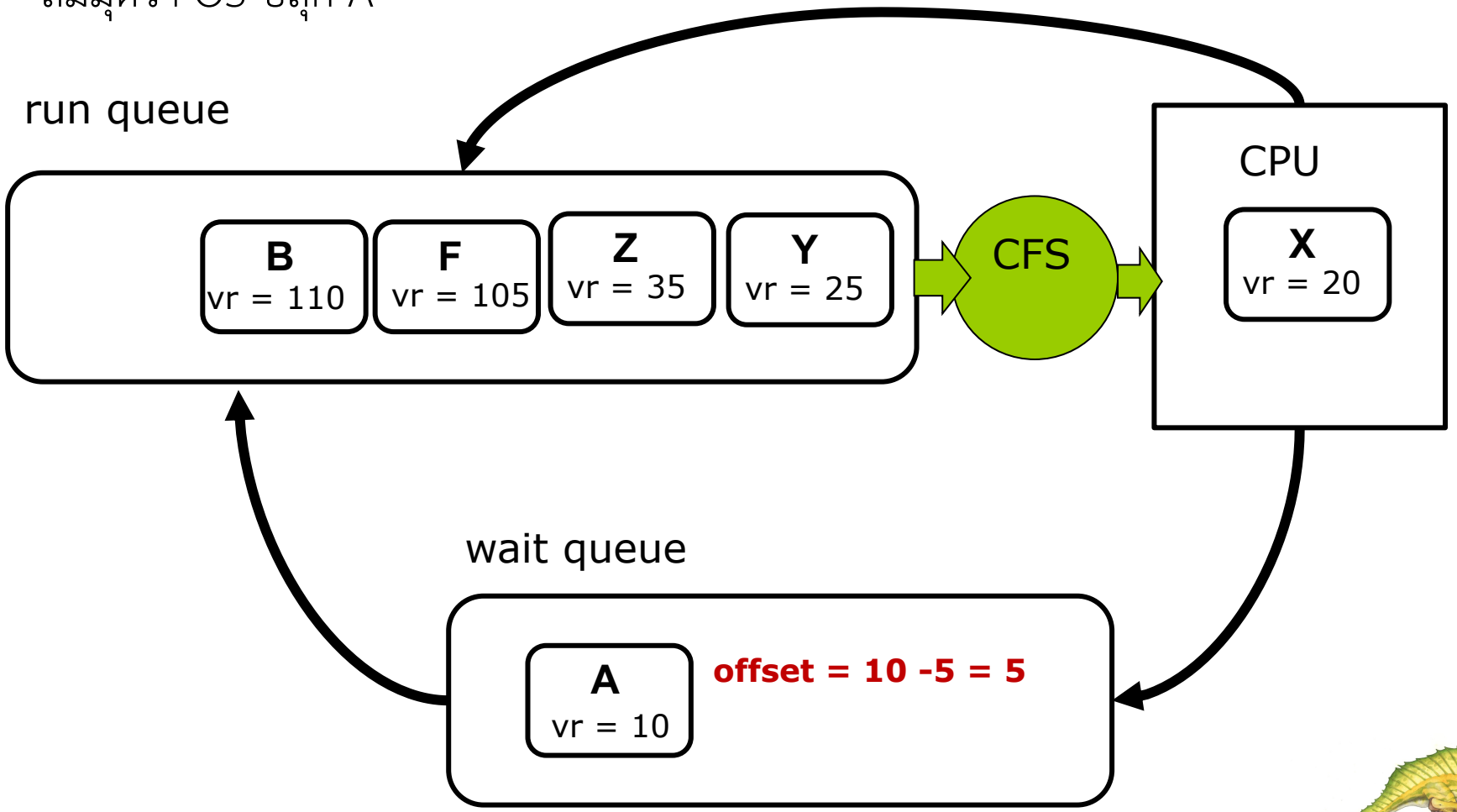
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- CFS จะคำนวณค่า offset ของ A คือ vr – Min vr = 10 – 5 = 5



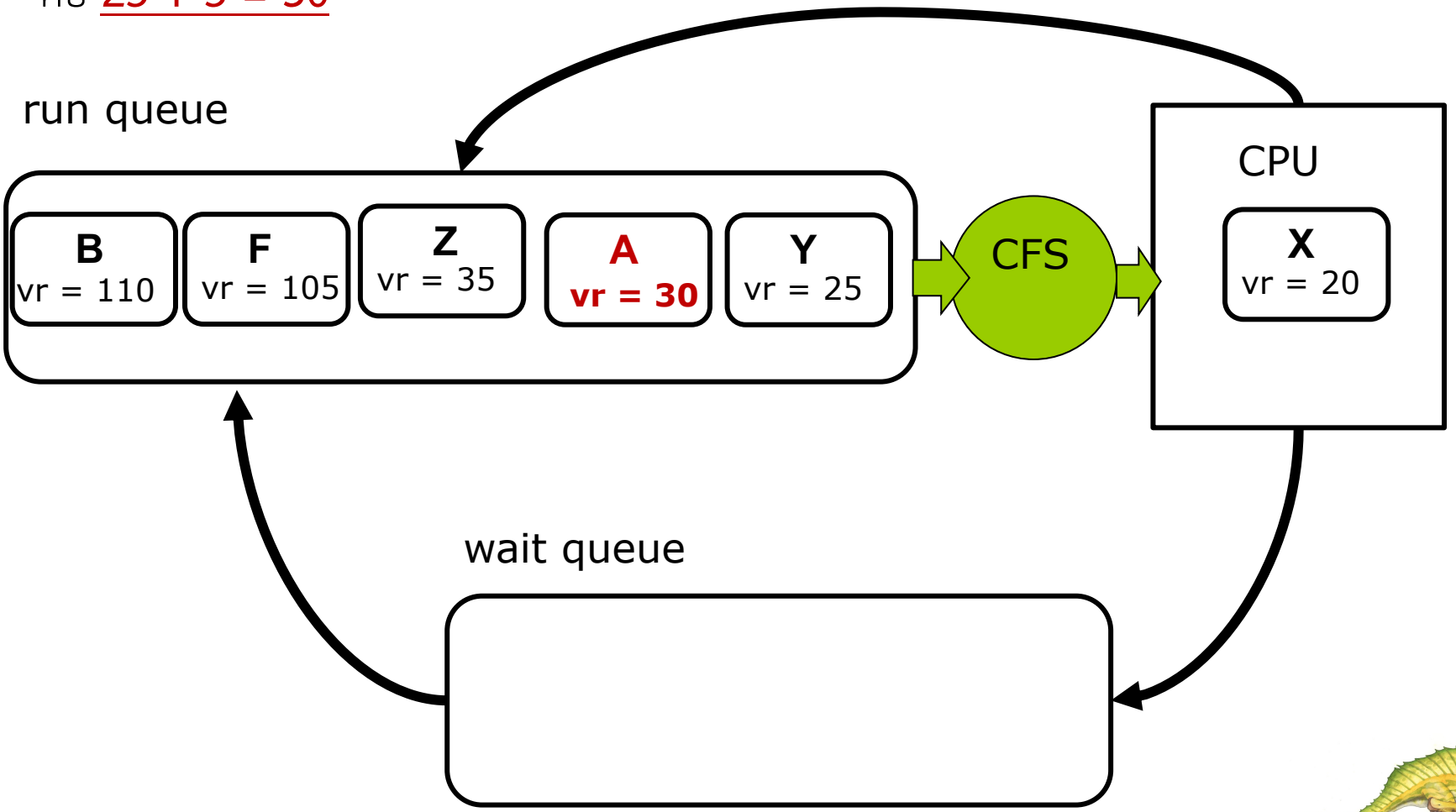
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- หลังจากนั้น 100 ms ผ่านไป run queue มีสภาพใหม่ดังนี้
- สมมุติว่า OS ปลุก A



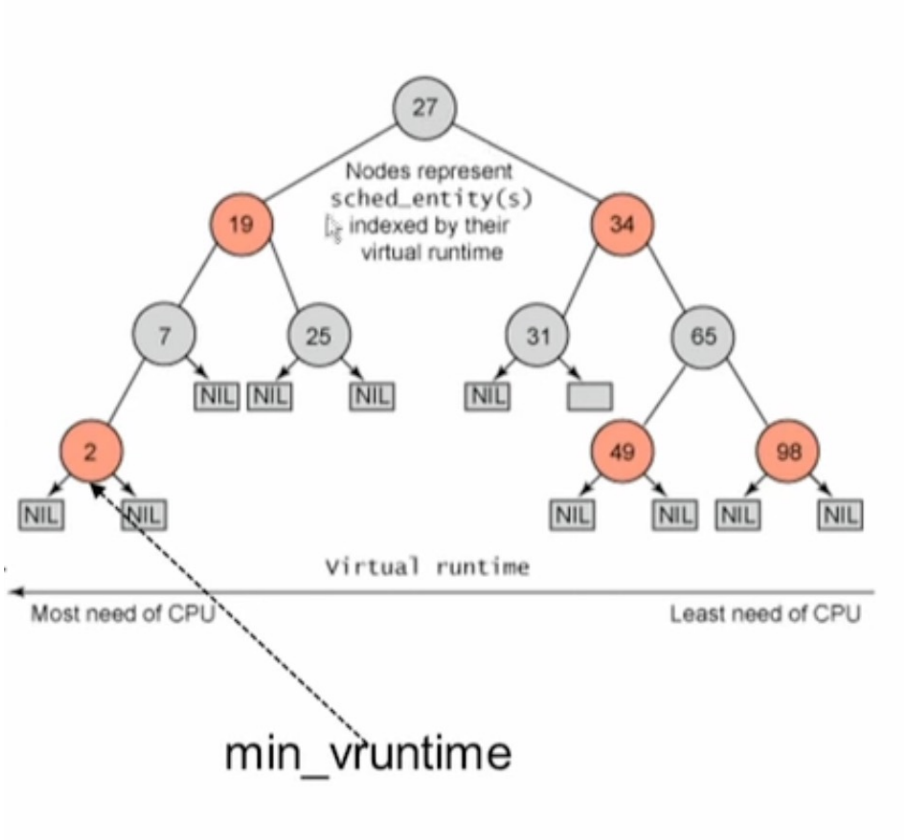
จำลอง CFS Scheduling ตย 3 สมมุติ TL=20

- CFS จะกำหนดให้ A มีค่า vruntime ใหม่ = Min vr ปัจจุบันของ Q + offset
- คือ 25 + 5 = 30



Run queue = Red-Black Tree

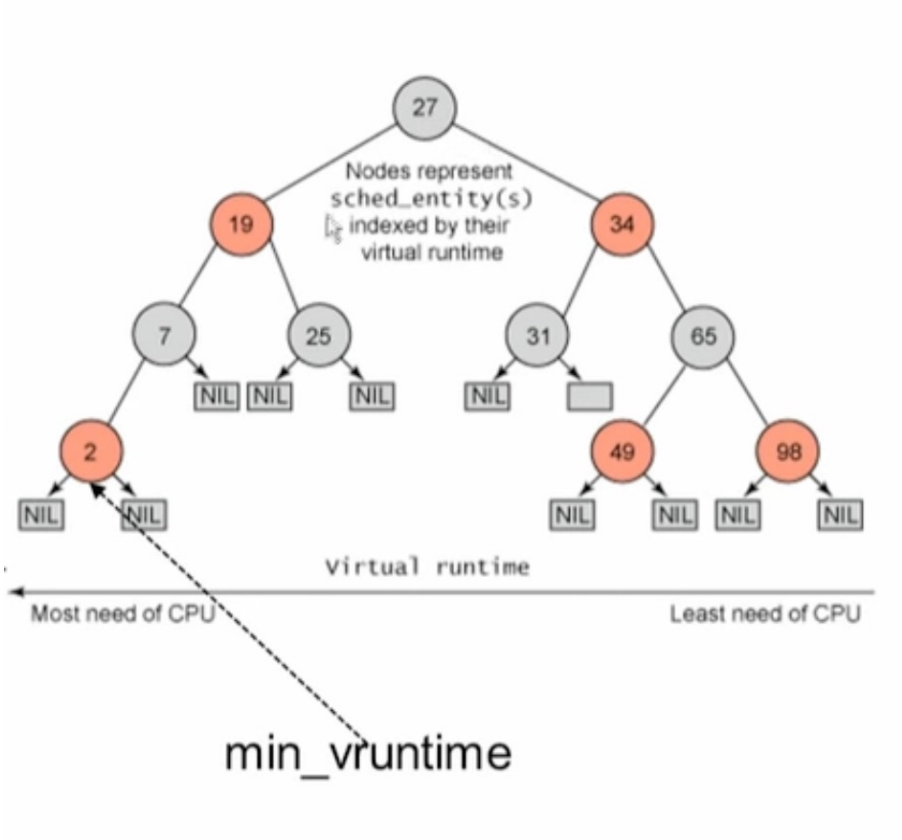
- CFS สร้าง Red-Black (RB) Tree เพื่อเก็บข้อมูลของ Runnable Process
  - RB Tree เป็น Balanced Binary Search Tree
  - https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- แต่ละ node ใน tree แทน a runnable task
- Node ทางด้านซ้ายจะมี vruntime ต่ำกว่า node ทางด้านขวาของ Tree
- Node ซ้ายสุด จะถูกชี้ด้วยตัวแปร **min_vruntime**



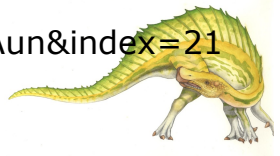
https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

Red-Black Tree

- เมื่อเกิด Context switching
  - CFS จะเลือก left-most node ของ RB Tree
  - ถูกชี้โดยตัวแปร min_vruntime
  - ใช้เวลา **O(1)**
- ถ้า Process ที่ใช้ CPU ก่อนหน้าเป็น runnable process
  - CFS จะนำ process มาใส่ใน RB-tree ตามค่า vruntime ของ process นั้น
  - ใช้เวลา **O(log n)**

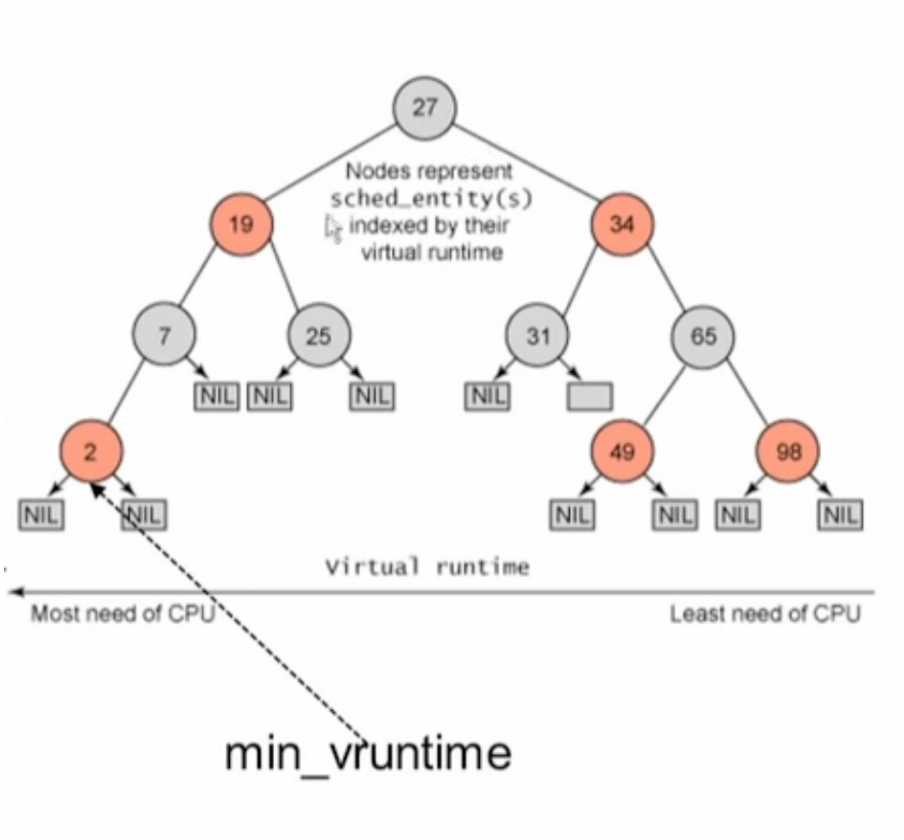


https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

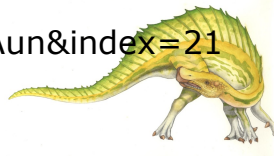


Red-Black Tree

- Process จะย้ายจากทางด้านซ้าย มาอยู่ทางด้านขวามือของ Tree เมื่อ execute และ CPU time เพิ่มขึ้นเรื่อยๆ
- จะไม่เกิดสถานการณ์ starvation (อดใช้งาน CPU) เพราะค่า vruntime ของ process จะเพิ่มขึ้นเสมอและจะมากกว่า process อื่นในที่สุด

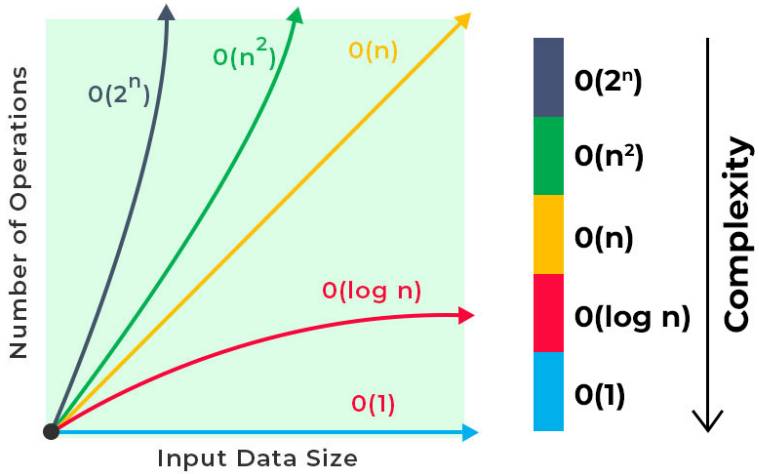


https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf



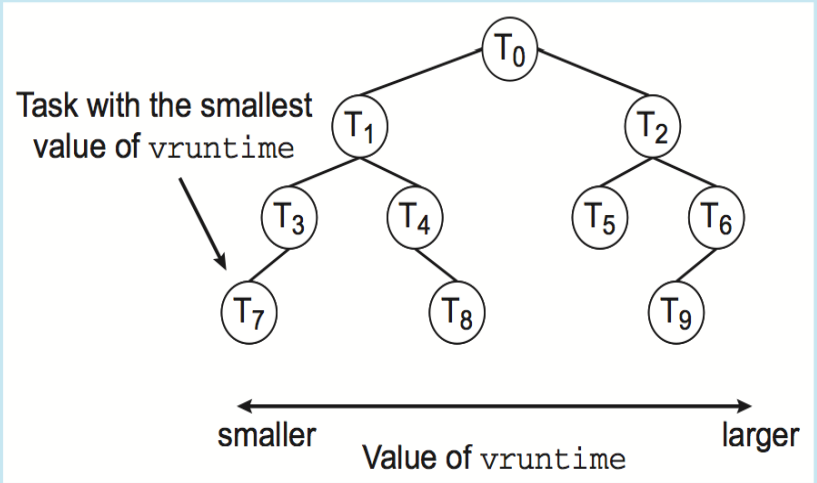
ประสิทธิภาพของ CFS

- การหา Task ที่มี vruntime น้อยที่สุดใน RB (Red Black) Tree ใช้เวลา O(log n) Time
- การหา node ที่มีค่าน้อยที่สุด การเพิ่ม node ใน Tree และการลบ node ออกจาก Tree ใช้เวลา O(log n) Time



https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/

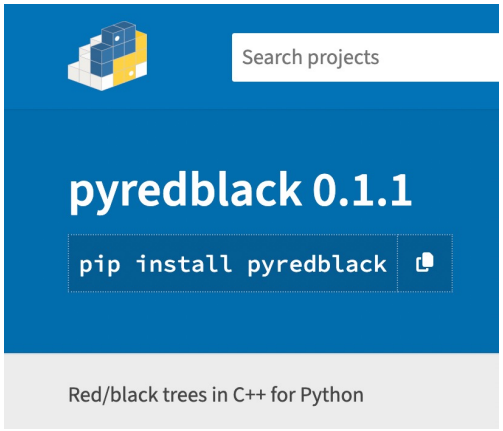
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Red-Black Tree (เพิ่มเติมสำหรับเขียนโปรแกรม)

- เป็น Binary Tree แบบ self balancing
- มีประสิทธิภาพสูงและมี Library ให้ใช้ในหลายภาษาโปรแกรม
- Python มี Library เช่น pyredblack
- https://pypi.org/project/pyredblack/
- หรือ C++ library เช่นที่
- https://github.com/niedong/stl_rbtree
- นศ สามารถดูการ self balancing ของ RB Tree ได้ที่
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html



https://www.youtube.com/watch?v=scfDOof9pww&list=PLEJxKK7AcSEGPOCFtQTJhOElU44J_JAun&index=21
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

หลักการ Completely Fair Scheduler 3

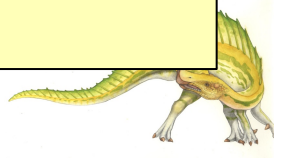
5. การสนับสนุนให้ I/O bound Task มี Responsiveness สูง

ข้อสังเกตุ:

5.1) CFS ไม่ต้อง Implement กระบวนการพิเศษ เพื่อสนับสนุนการตอบสนอง ของ I/O bound Tasks

5.2) จากวิธีการจัดการ vruntime ของ Task ที่ใช้ I/O ที่ได้กล่าวถึงก่อนหน้า CFS จะให้โอกาส I/O bound Task ที่เพิ่งทำ I/O เสร็จ มีสิทธิ์ถูกเลือกเท่ากับ Task ที่มีโอกาสที่จะถูกเลือกเข้ามาใช้งาน CPU มากที่สุด เท่านั้น *มันจะไม่มีอภิสิทธิ์มากกว่า Tasks ที่รออยู่แล้ว*

5.3) ถึงแม้จะถูกนำกลับมาไว้ที่ต้นคิว I/O bound Task จะไม่ทำให้เกิด starvation เพราะ I/O bound task จะเรียกใช้ I/O บ่อย ทำให้ CPU bound Task ได้เข้ามาใช้ CPU



6. ทุก Process ได้รับเวลาเสมือนที่แบ่งจาก CPU เท่ากัน
   แต่เวลาจริงที่จะรัน ขึ้นอยู่กับ Static Priority ของแต่ละ Process
7. CFS ไม่มี dynamic priority คือไม่เปลียน priority ขณะ process รัน
8. CFS ต้องการให้ priority ในระดับที่ต่างกัน 1 อันดับส่งผลให้
   Time Slice ต่างกันประมาณ 10% (uniform)
9. หลังจาก process ได้รันใน CPU ตามเวลา Time Slice ที่ scheduler กำหนด
   Process อาจถูกขัดจังหวะก่อน Time slice หมดได้ ถ้ามี Process ที่ vruntime ต่ำกว่าเข้ามาใน runqueue

nice	Time Slice
0	T0 +10%
-1	T(-1) +10%
-2	T(-2) +10%
-3	T(-3)
....	
-20	T(-20) +10%

nice	Time Slice
0	T0 -10%
1	T1 -10%
2	T2 -10%
3	T3
....	
19	T19 -10%

สถานการณ์เปรียบเทียบ

1. สมมุติว่า นักกีฬาซูโม่เช่ารถบัส เดินทางไปสนามกีฬา ด้วยราคาเหมาจ่าย T และ

2. สมาคมกีฬาซูโม่กำหนดให้นักกีฬา **ซูโม่** มีระดับของน้ำหนัก **40 ระดับ ดังนี้**

อ้างอิง: created by Bing AI

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

3. กำหนดให้ ศัพท์เฉพาะของซูโม่เรียกระดับ นน แต่ละระดับว่า nice และเราแปลง nice เป็นค่า นน จริงด้วยสูตร weight = sched_prio_to_weight[nice+20]

4. นน ปกติ คือค่า weight ที่ได้จากค่า nice = 0 ซึ่งจะเท่ากับ weight = sched_prio_to_weight[0+20] = 1024 kg (ต้องตัวใหญ่มาก)

สถานการณ์เปรียบเทียบ 1

1. **สมมุติว่า นักกีฬาซูโม่เช่ารถบัส เดินทางไปสนามกีฬา ด้วยราคาเหมาจ่าย T** และนักกีฬาตกลงกันว่าจะแชร์กันจ่าย โดยที่**แต่ละคนจะจ่ายตาม นำหนัก (weight) ตัวของตน**

อ้างอิง: created by Bing AI

2. **สมมุติว่ามี ซูโม่ n คนแต่ละคนมี weight เท่ากับ**

$$1024,1277,1024,820$$

นำหนักรวม (Total weight หรือ tw) จะเท่ากับ

$$tw = 4145$$

3. สัดส่วนของ นน ของ ผู้โดยสารแต่ละคนต่อ นน ทั้งหมด จะเท่ากับ

$$\frac{1024}{4145}, \frac{1277}{4145}, \frac{1024}{4145}, \frac{820}{4145}$$ หรือประมาณ $0.25,0.30,0.25,0.20$

รวมกันเท่ากับ 1

สถานการณ์เปรียบเทียบ 1

4. ดังนั้น ซูโม่คนที่ 1 จะจ่าย เงินเท่ากับ $t_1 = 0.25 \times T$

ซูโม่คนที่ 2 จะจ่าย เงินเท่ากับ $t_2 = 0.3 \times T$

ซูโม่คนที่ 3 จะจ่าย เงินเท่ากับ $t_3 = 0.25 \times T$

ซูโม่คนที่ 4 จะจ่าย เงินเท่ากับ $t_4 = 0.2 \times T$

5. แต่เลขาซูโม่จดว่า

ซูโม่ 2 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 2 จ่าย เงินเท่ากับ $tv_2 = 0.25 \times T$ ทั้งๆที่จริงๆจ่าย $0.3 \times T$ (จ่ายจริงมาก แต่จดว่าปกติ)

ซูโม่ 4 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 4 จ่าย เงินเท่ากับ $tv_4 = 0.25 \times T$ ทั้งๆที่จริงๆจ่าย $0.2 \times T$ (จ่ายจริงน้อย แต่จดว่าปกติ)

สมมุติว่า ซูโม่ที่เลขาจดว่าจ่ายน้อย จะได้คิวทานอาหารก่อน (ยิ่งน้อยยิ่งดี)

เปรียบเหมือนการที่ scheduler เลือก process เข้าใช้ cpu ในอนาคต

จงใจทำให้คนมี priority มี vruntime ต่ำกว่าที่ใช้จริงและเพิ่มโอกาสที่จะถูกเลือกรอบหน้า

High prio => จดน้อยกว่าที่ใช้จริง ส่วน low prio => จดมากกว่าที่ใช้จริง

สถานะการณ์เปรียบเทียบ 2

1. **สมมุติว่า นักกีฬาซูโม่เช่ารถบัส เดินทางไปสนามกีฬา ด้วยราคาเหมาจ่าย T** และนักกีฬาตกลงกันว่าจะแชร์กันจ่าย โดยที่**แต่ละคนจะจ่ายตาม นำหนัก (weight) ตัวของตน**

อ้างอิง: created by Bing AI

2. **สมมุติว่ามี ซูโม่ n คนแต่ละคนมี weight เท่ากับ**

$$1024,655,526,820$$

นำหนักรวม (Total weight หรือ tw) จะเท่ากับ

$$tw = 3025$$

3. สัดส่วนของ นน ของ ผู้โดยสารแต่ละคนต่อ นน ทั้งหมด จะเท่ากับ

$$\frac{1024}{3025}, \frac{655}{3025}, \frac{526}{3025}, \frac{820}{3025}$$ หรือประมาณ $0.34,0.22,0.17,0.27$

รวมกันเท่ากับ 1

สถานการณ์เปรียบเทียบ 2

4. ดังนั้น ซูโม่คนที่ 1 จะจ่าย เงินเท่ากับ $t_1 = 0.34 \times T$

ซูโม่คนที่ 2 จะจ่าย เงินเท่ากับ $t_2 = 0.22 \times T$

ซูโม่คนที่ 3 จะจ่าย เงินเท่ากับ $t_3 = 0.17 \times T$

ซูโม่คนที่ 4 จะจ่าย เงินเท่ากับ $t_4 = 0.27 \times T$

5. แต่เลขาซูโม่จดว่า

ซูโม่ 2 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 2 จ่าย เงินเท่ากับ $tv_2 = 0.34 \times T$ ทั้งๆที่จริงๆจ่าย $0.22 \times T$ (จ่ายจริงน้อย แต่จดว่าปกติ)

ซูโม่ 3 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 4 จ่าย เงินเท่ากับ $tv_3 = 0.34 \times T$ ทั้งๆที่จริงๆจ่าย $0.17 \times T$ (จ่ายจริงน้อย แต่จดว่าปกติ)

ซูโม่ 4 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 4 จ่าย เงินเท่ากับ $tv_4 = 0.34 \times T$ ทั้งๆที่จริงๆจ่าย $0.27 \times T$ (จ่ายจริงน้อย แต่จดว่าปกติ)

สมมุติว่า ซูโม่ที่เลขาจดว่าจ่ายน้อย จะได้คิวทานอาหารก่อน (ยิ่งน้อยยิ่งดี)

เปรียบเหมือนการที่ scheduler เลือก process เข้าใช้ cpu ในอนาคต

จงใจทำให้คนมี priority มี vruntime ต่ำกว่าที่ใช้จริงและเพิ่มโอกาสที่จะถูกเลือกรอบหน้า

High prio => จดน้อยกว่าที่ใช้จริง ส่วน low prio => จดมากกว่าที่ใช้จริง

สถานการณ์เปรียบเทียบ 3

1. **สมมุติว่า นักกีฬาซูโม่เช่ารถบัส เดินทางไปสนามกีฬา ด้วยราคา T** และนักกีฬาตกลงกันว่าจะแชร์กันจ่าย โดยที่ **แต่ละคนจะจ่ายตาม นำหนัก (weight) ตัวของตน**

2. **สมมุติว่ามี ซูโม่ n คนแต่ละคนมี weight เท่ากับ**

$$1024,3906,1991,820$$

นำหนักรวม (Total weight หรือ tw) จะเท่ากับ

$$tw = 7741$$

อ้างอิง: created by Bing AI

3. สัดส่วนของ นน ของ ผู้โดยสารแต่ละคนต่อ นน ทั้งหมด จะเท่ากับ

$\frac{1024}{7741}, \frac{3906}{7741}, \frac{1991}{7741}, \frac{820}{7741}$ หรือประมาณ $0.13,0.50,0.26,0.11$

รวมกันเท่ากับ 1

สถานะการณ์เปรียบเทียบ 3

4. ดังนั้น ซูโม่คนที่ 1 จะจ่าย เงินเท่ากับ $t_1 = 0.13 \times T$

ซูโม่คนที่ 2 จะจ่าย เงินเท่ากับ $t_2 = 0.50 \times T$

ซูโม่คนที่ 3 จะจ่าย เงินเท่ากับ $t_3 = 0.26 \times T$

ซูโม่คนที่ 4 จะจ่าย เงินเท่ากับ $t_4 = 0.11 \times T$

5. แต่เลขาซูโม่จดว่า

ซูโม่ 2 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 2 จ่าย เงินเท่ากับ $tv_2 = 0.13 \times T$ ทั้งๆที่จริงๆจ่าย $0.50 \times T$ (จ่ายจริงมาก แต่จดว่าปกติ)

ซูโม่ 3 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 4 จ่าย เงินเท่ากับ $tv_3 = 0.13 \times T$ ทั้งๆที่จริงๆจ่าย $0.26 \times T$ (จ่ายจริงมาก แต่จดว่าปกติ)

ซูโม่ 4 มี นน คือ 1024 และจะจดไว้ว่าซูโม่คนที่ 4 จ่าย เงินเท่ากับ $tv_4 = 0.13 \times T$ ทั้งๆที่จริงๆจ่าย $0.11 \times T$ (จ่ายจริงน้อย แต่จดว่าปกติ)

สมมุติว่า ซูโม่ที่เลขาจดว่าจ่ายน้อย จะได้คิวทานอาหารก่อน (ยิ่งน้อยยิ่งดี)

เปรียบเหมือนการที่ scheduler เลือก process เข้าใช้ cpu ในอนาคต

จงใจทำให้คนมี priority มี vruntime ต่ำกว่าที่ใช้จริงและเพิ่มโอกาสที่จะถูกเลือกรอบหน้า

High prio => จดน้อยกว่าที่ใช้จริง ส่วน low prio => จดมากกว่าที่ใช้จริง

กฎ CPU Scheduling part 3

1. **สมมุติว่า นักกีฬาซูโม่เช่ารถบัส เดินทางไปสนามกีฬา ด้วยราคาเหมาจ่าย T** และนักกีฬาตกลงกันว่าจะแชร์กันจ่าย โดยที่**แต่ละคนจะจ่ายตาม นำหนัก (weight) ตัวของตน**

2. **สมมุติว่ามี ซูโม่ n คนแต่ละคนมี weight เท่ากับ**

$$w_1, w_2, w_3, \dots, w_n$$

นำหนักรวม (Total weight หรือ tw) จะเท่ากับ

$$tw = \sum_{i=1}^{n} w_i$$

อ้างอิง: created by Bing AI

3. สัดส่วนของ นน ของ ผู้โดยสารแต่ละคนต่อ นน ทั้งหมด จะเท่ากับ $\frac{w_1}{tw}, \frac{w_2}{tw}, \dots, \frac{w_n}{tw}$

รวมกันเท่ากับ 1

4. **ดังนั้น ซูโม่คนที่ i จะจ่ายเงินจริงเท่ากับ $t_i = \frac{w_i}{tw} \times T$**

5. สมมุติ **นน ปกติคือ $w = 1024$** เลขาจดว่าซูโม่คนที่ i จ่ายเงิน

$$tv_i = \frac{w}{tw} \times T$$ แทนที่จะเป็น $t_i = \frac{w_i}{tw} \times T$

อ้างอิง https://helix979.github.io/jkoo/post/os-scheduler/
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html

การกำหนดค่า Time Slice

CFS กำหนดค่า Time Slice จริงที่จะให้ CPU core รัน Task i

- กำหนดให้ มี Task n Task ใน run queue และแต่ละ Task มี Priority ของตนเองคือ $w_1, w_2, w_3, \ldots, w_n$ และ $tw = \sum_{i=1}^{n} w_i$ ค่า Time Slice ของ Task i เท่ากับ

สัดส่วนของ weight มีอิทธิพลโดยตรงต่อ Time Slice

$$ts_i = \frac{w_i}{tw} \times T \qquad \text{EQ1}$$

โดยที่ค่า ts_i คือค่า Time Slice จริงที่ CFS จะให้ Task รันบน CPU และ T คือค่า Target Latency

อ้างอิง https://helix979.github.io/jkoo/post/os-scheduler/
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html
https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-nice/

สูตรที่มาของตาราง Weight

- CFS กำหนดให้มีระดับของ weight 40 ระดับ ดังนี้

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

- เราระดับ นน แต่ละระดับว่า nice และเราแปลงค่า nice เป็นค่า weight ด้วย

  weight = sched_prio_to_weight [nice+20]

- weight ปกติ คือค่า weight ที่ได้จากค่า nice = 0 ซึ่งจะเท่ากับ 1024

  weight = sched_prio_to_weight[0+20] = 1024

- ค่าในตารางที่มีค่า *nice* เป็นดัชนีได้มาจาก: weight = $1024/1.25^{nice}$

อ้างอิง https://helix979.github.io/jkoo/post/os-scheduler/
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html
https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-nice/

การเพิ่มค่า vruntime

การกำหนดค่า vruntime หลังจาก Task i ประมวลผลเป็นเวลา t_i

- หลังจาก process ได้รันใน CPU ตามเวลา Time Slice ts_i ที่ scheduler กำหนด Process อาจถูกขัดจังหวะก่อน Time slice หมดได้ ถ้ามี Process ที่ vruntime ต่ำกว่าเข้ามาใน runqueue
- สมมุติว่า CPU ประมวลผล Task i เป็นเวลา t_i (ซึ่ง $t_i \leq ts_i$) เมื่อ CFS นำ Task i ส่งกลับไปที่ run queue มันจะนำค่า t_i มาคำนวณหาค่า vruntime ดังนี้

$$vruntime_i \mathrel{+}= t_i \times \frac{1024}{w_i} \qquad \textbf{EQ2}$$

ทำไม CFS ใช้ Equation **EQ2** นี้

อ้างอิง https://helix979.github.io/jkoo/post/os-scheduler/
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html
https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-nice/

การกำหนดค่า vruntime

ข้อสังเกต:

- เมื่อ Task i ประมวลผลเป็นเวลา t_i ก่อนถูกส่งกลับไปที่ run queue มันจะคำนวณค่า vruntime ดังนี้

$$vruntime_i \mathrel{+}= t_i \times \frac{1024}{w_i}$$

- พิสูจน์ว่าสูตรนี้คือการ normalize ค่า t_i ให้อยู่ภายใต้ **normal weight** สมมุติว่า $t_i = ts_i$ แทนค่า t_i ด้วย EQ1 เราจะได้ ค่า ts_i กรณี weight ปกติ

$$vruntime_i \mathrel{+}= \left(\frac{w_i}{tw}\right) \times T \times \frac{1024}{w_i}$$

$$vruntime_i \mathrel{+}= \left(\frac{1024}{tw}\right) \times T$$

อ้างอิง https://helix979.github.io/jkoo/post/os-scheduler/
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html
https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-nice/

กฎ CPU Scheduling part 3

ข้อสังเกต:

- เนื่องจาก weight ปกติคือ $w = 1024$ และเราได้

$$\boldsymbol{vruntime_i} \mathrel{+}= \left(\frac{\mathbf{1024}}{\boldsymbol{tw}}\right) \times \boldsymbol{t_i}$$

- ดังนั้น ไม่ว่าค่าเวลา $\boldsymbol{t_i}$ ของ Task จะมีค่า Priority เท่าใดก็ตาม CFS จะ normalize $\boldsymbol{t_i}$ ให้เป็นค่าเวลา Time Slice เมื่อ nice value เป็นปกติ (เท่ากับ 0) หรือ weight = 1024 เสมอ ก่อนที่มันจะส่ง Task ไปไว้ใน run queue
- เป็นการทำให้ Process ที่มี priority สูงมีโอกาสเข้ามาใช้ cpu มากกว่า process priority ต่ำในรอบหน้า

จำลอง CPU Scheduling ตย 2 สมมุติ TL=20

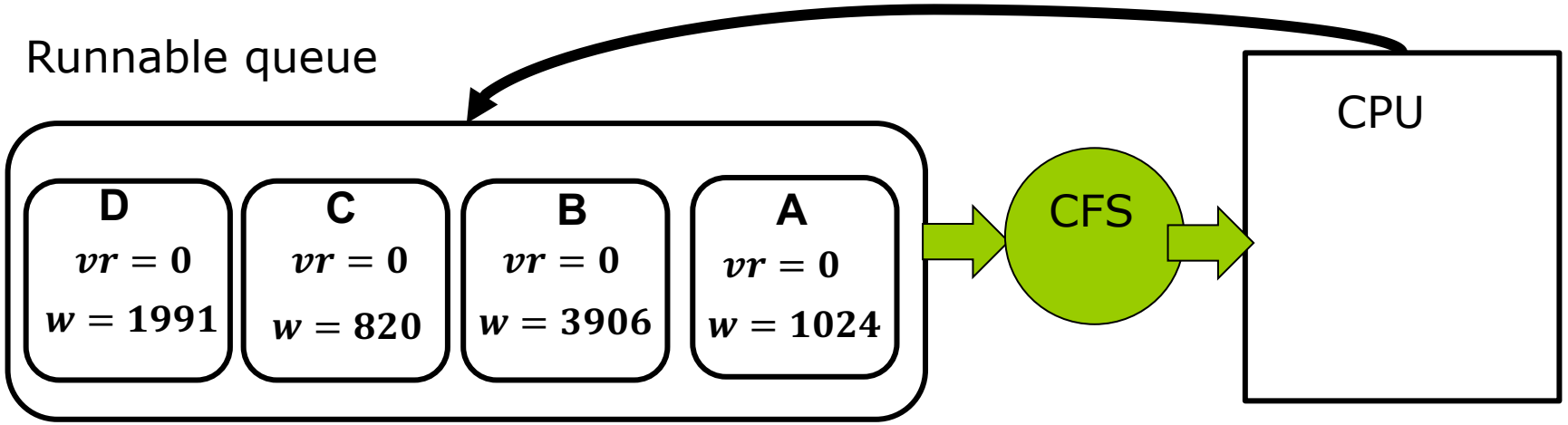
- เราจะแสดงตัวอย่างการจำลอง CFS CPU scheduling อีกครั้ง
- แต่คราวนี้เราจะเพิ่มการ สร้าง (fork) task ใหม่เข้าสู่ run queue
- ข้อกำหนด
  - มีค่า TL = 20
  - มี Tasks (อาจเป็น Process หรือ Thread) เริ่มต้น 4 Tasks
  - สมมุติว่าค่า **vruntime** (หรือ vr) เริ่มต้นของทุก Task คือ $vr = 0$
  - มี Task Nice value และ Weight ดังนี้

TASK	A	B	C	D	Total W
NICE	0	−6	1	−3	
Weight	1024	3906	820	1991	7741

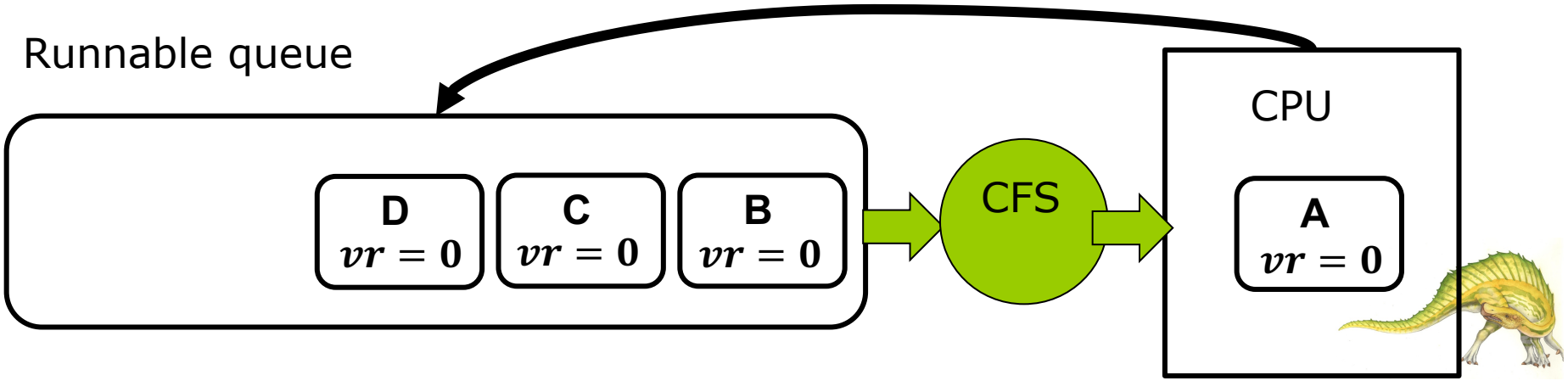
จำลอง CFS Scheduling สมมุติ TL=20

- เริ่มต้น N = 4 (เพื่อความง่ายเราสมมุติว่ารันหมด time slice)

$w = 102$

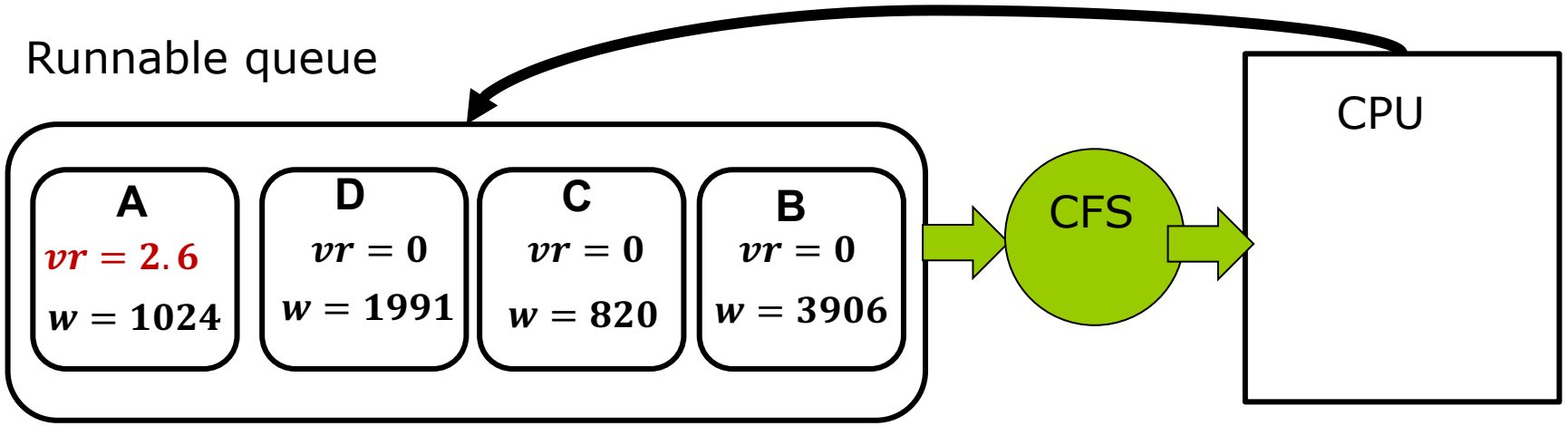


- เลือก A คำนวณค่า time_slice ของ A คือ $ts_A = \frac{1024}{7741} \times 20 = 0.13 \times 20 = 2.6$

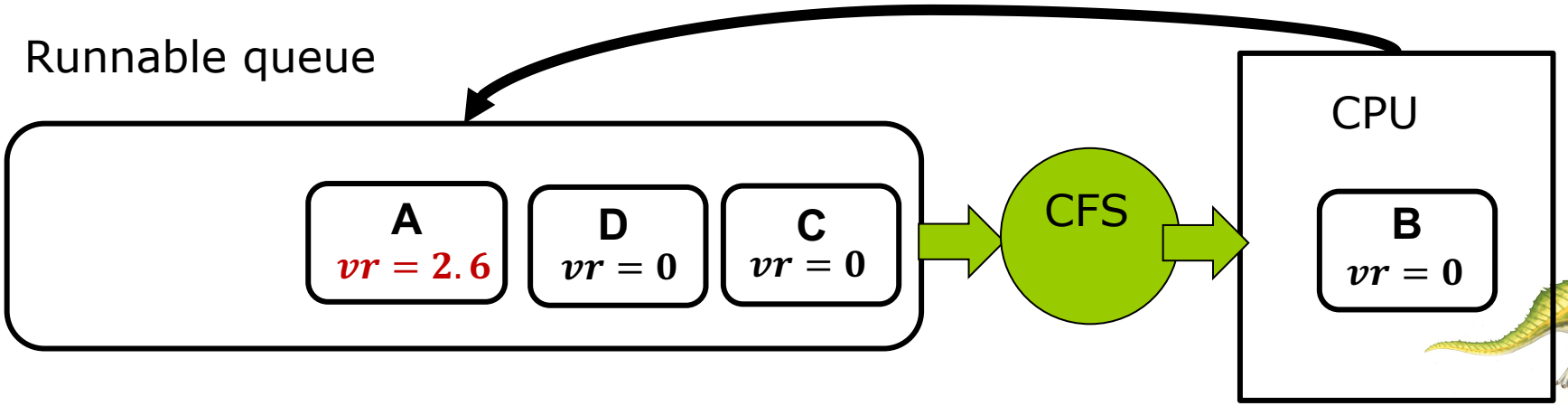


จำลอง CFS Scheduling สมมุติ TL=20

- เมื่อ A จบ Time Slice CFS คำนวณ $vr_A \mathrel{+}= 2.6 \times \frac{1024}{1024} = 2.6$

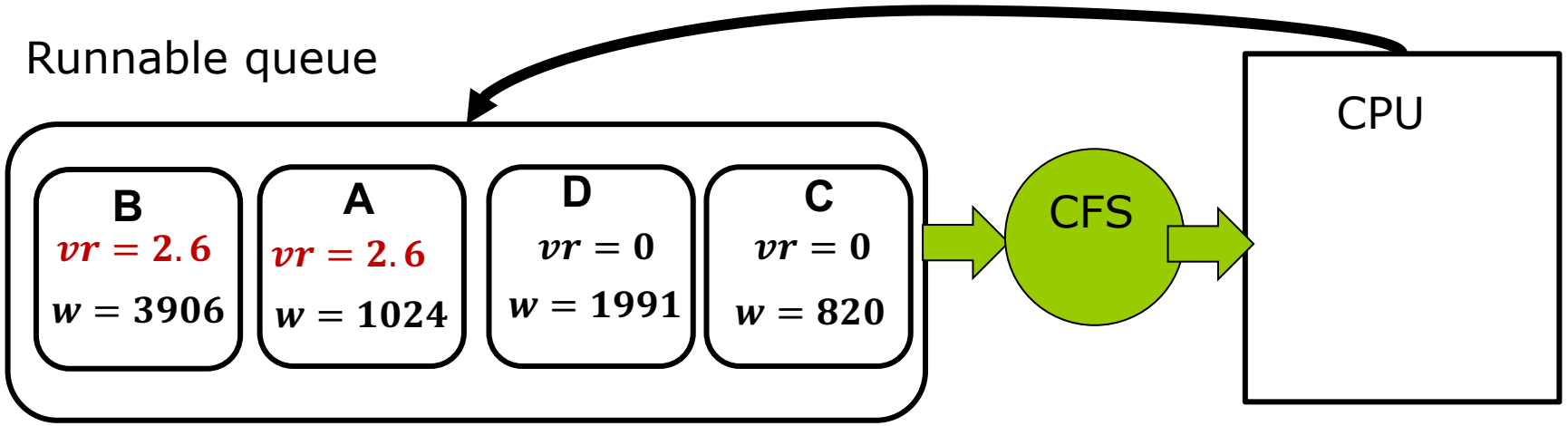


- เลือก B คำนวณค่า time_slice ของ B คือ $ts_B = \frac{3906}{7741} \times 20 = 0.5 \times 20 = 10$

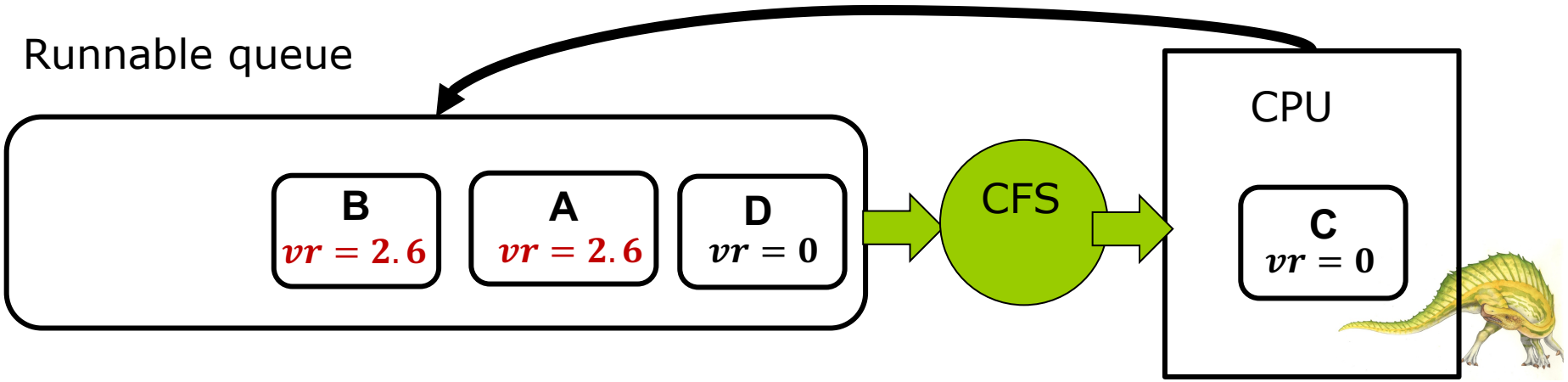


จำลอง CFS Scheduling สมมุติ TL=20

- เมื่อ B จบ Time Slice CFS คำนวณ $vr_B \mathrel{+}= 10 \times \frac{1024}{3906} \approx 2.6$

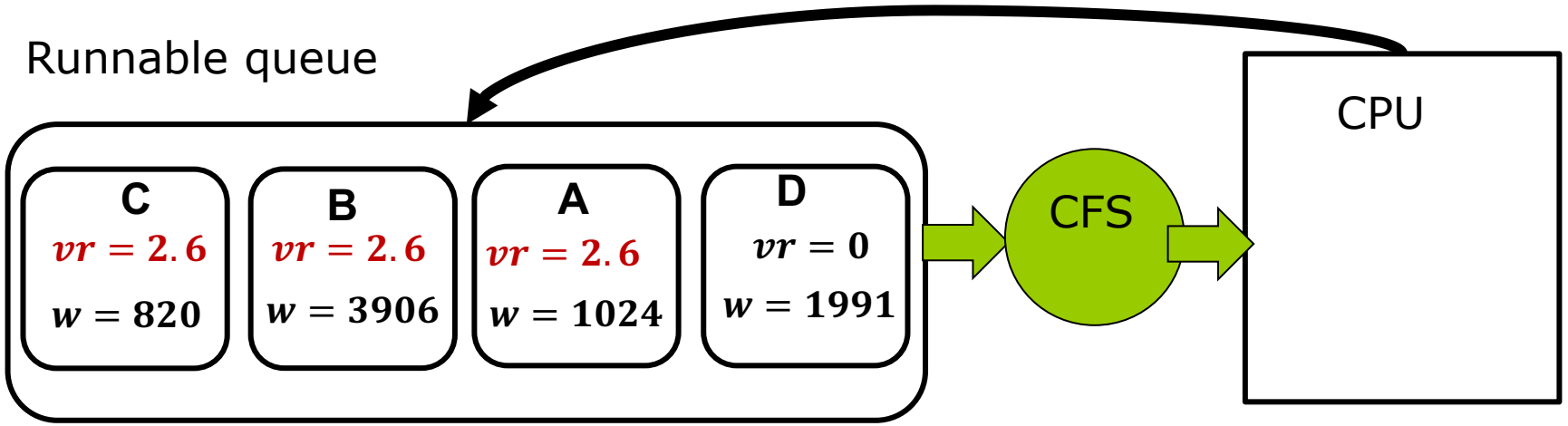


- เลือก C คำนวณค่า time_slice ของ C คือ $ts_C = \frac{820}{7741} \times 20 = 0.11 \times 20 = 2.2$

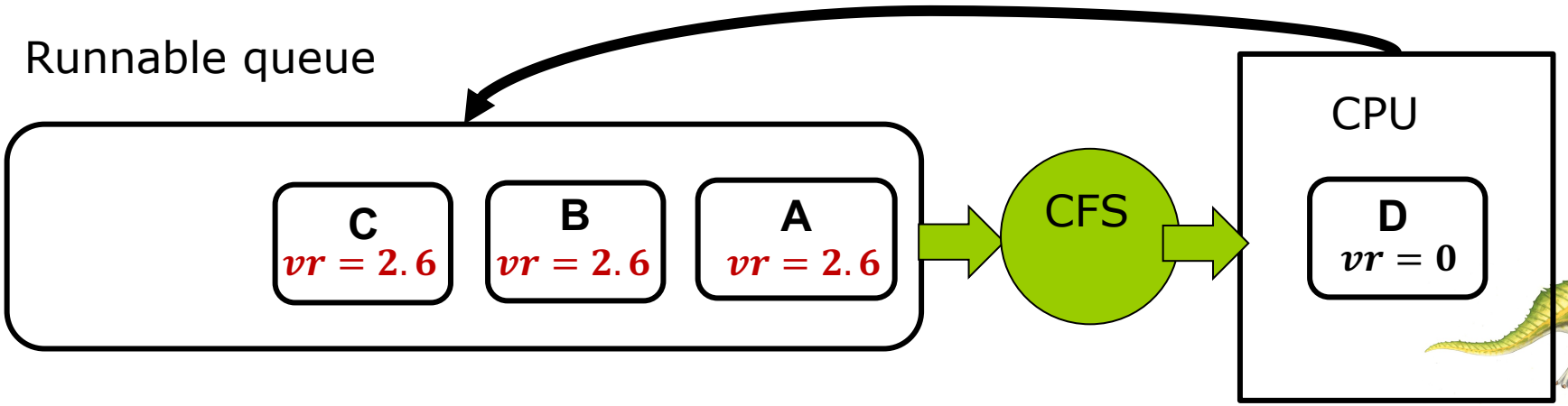


จำลอง CFS Scheduling สมมุติ TL=20

- เมื่อ C จบ Time Slice CFS คำนวณ $vr_C \mathrel{+}= 2.2 \times \frac{1024}{820} \approx 2.6$

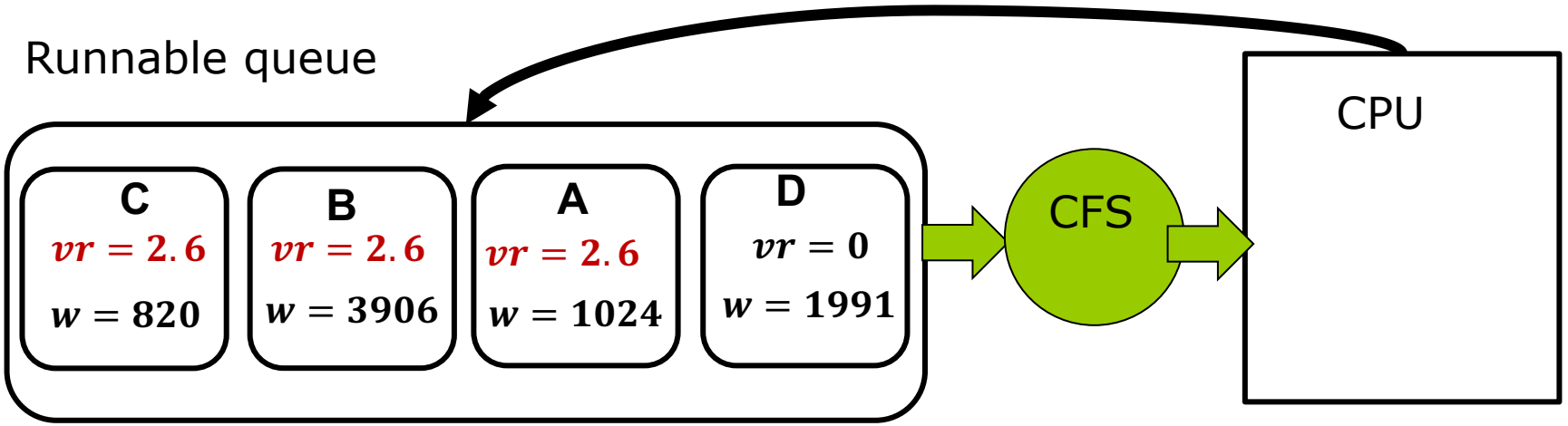


- เลือก D คำนวณค่า time_slice ของ C คือ (การบ้าน) ts_D =?

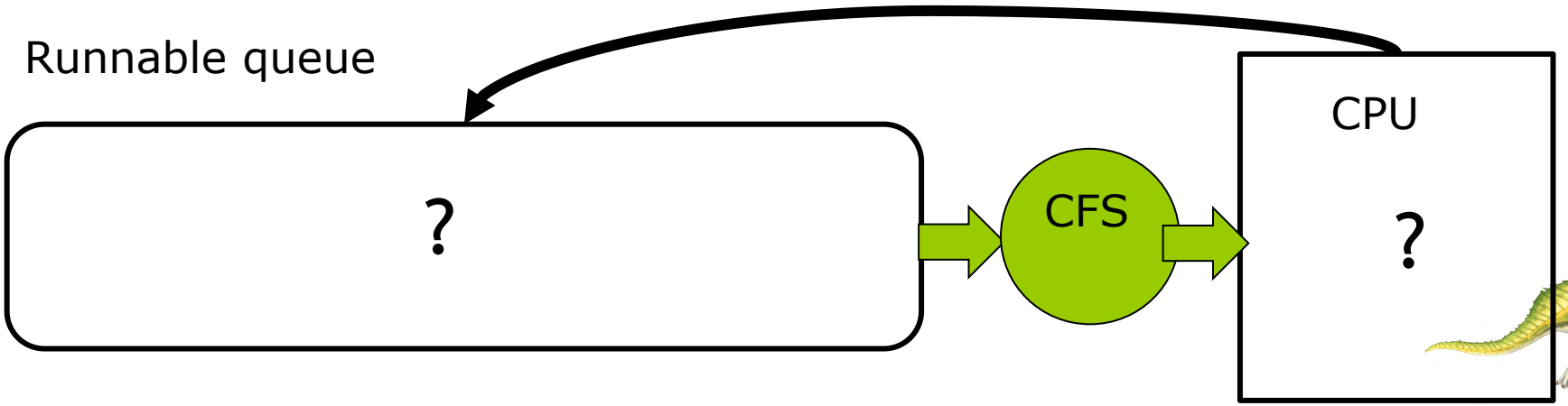


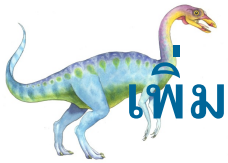
จำลอง CFS Scheduling สมมุติ TL=20

- เมื่อ D จบ Time Slice CFS คำนวณ vr_D +=?



- เลือก ? คำนวณค่า time_slice ของ ? คือ (การบ้าน)





เพิ่มความ Fair ด้วย CFS Group Scheduling

- ในกรณีที่ ผู้ใช้ user มีหลายคน เช่น User A และ User B ในระบบ ผู้ใช้แต่ละคนอาจมีจำนวน Tasks จำนวน ไม่เท่ากัน
- เนื่องจากทุก Task ได้สัดส่วนของเวลาเท่ากันคือ 1/N ดังนั้น ผู้ใช้ที่มี Task มากกว่าก็จะได้รับเวลารวมของการประมวลผลบน CPU มากกว่า
- CFS อนุญาตให้มีการกำหนด Group Scheduling สมมุติว่า User A มี 30 Tasks แต่ User B มี 10 Tasks
- สมมุติว่าเราแบง Tasks เป็น 2 Groups แต่ละกลุ่มจะได้รับเวลา 50%
- CFS จะแบ่งเวลา 50% ของ User A ให้ 30 Tasks ของ Group ที่ 1 และ
- แบ่งเวลา 50% ของ User B ให้ 10 Tasks ในกลุ่มที่ 2

http://yajin.org
https://mechpen.github.io/posts/2020-04-27-cfs-group/index.html

Linux Scheduling in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (Cont.)

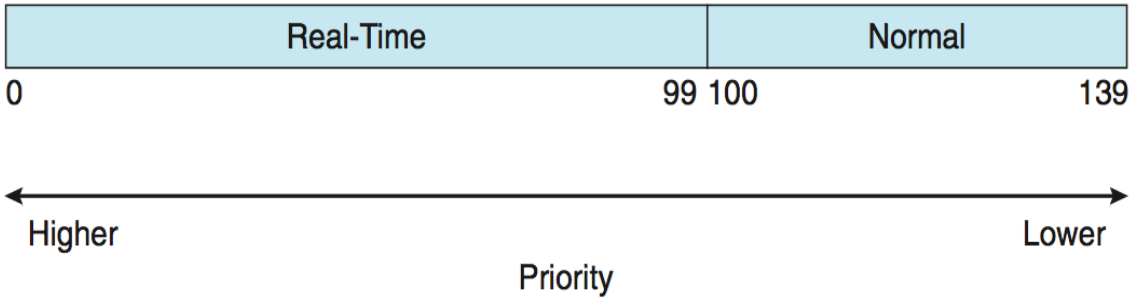
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

สรุป หลักการ CFS

1. CFS แบ่งเวลาของ CPU ให้ยุติธรรม (Fair) มี N Tasks ก็ได้ 1/N ของ TL
2. Task จะสะสมเวลาที่ใช้ CPU ใน vruntime และใช้มันเป็นเครื่องมือจัดอันดับ
3. CFS เลือกรัน Task ใน run queue ที่ได้ vruntime น้อยที่สุดก่อน
4. Task ทีเกิดใหม่จะมี vruntime เท่ากับของ Task ที่มีเวลา vruntime น้อยที่สุดใน run queue ของ cpu core ที่มันถูกส่งไปรัน
5. Task ที่รอ I/O หรือหลับ (รอ ใน wait queue) จะมี vruntime เท่ากับของ Task ที่มีเวลา vruntime น้อยที่สุดใน run queue ของ local cpu core
6. Task ที่ถูกนำออกจาก run queue ไปรอ I/O เมื่อกลับมารอ CPU อีกที ต้องกลับมารอที่ คิวอันดับเดิมเท่ากับตอนก่อนออกไป
7. ทุก Process ได้รับ vruntime เท่ากัน แต่เวลาที่จะรันบน CPU จริงๆ นั้นเป็นอีกเรื่องหนึ่ง แยกออกจากการเลือก Process ของ Scheduler
8. เวลาที่ Process รันจริงขึ้นอยู่กับ Priority ของแต่ละ Process

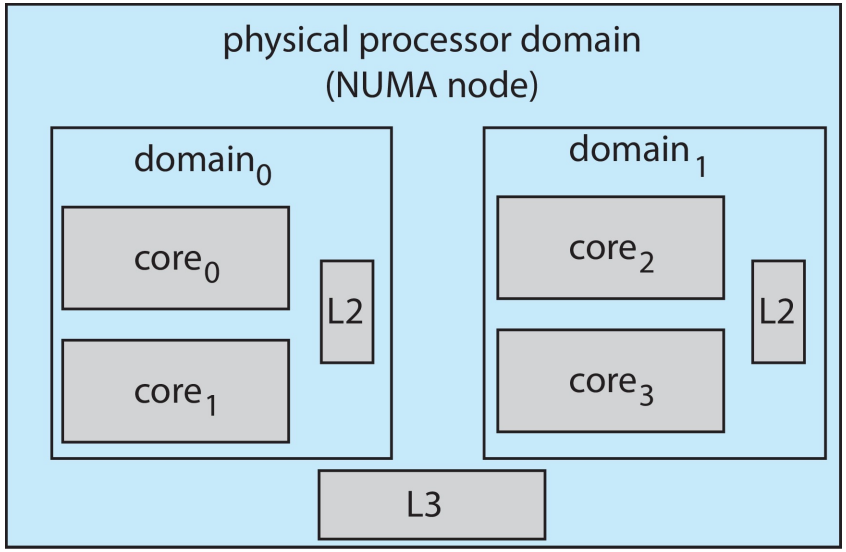
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.



สรุป

- Linux Scheduler
- หลักการของ CFS scheduling
- ขั้นตอนของ CFS
- Task Priority ใน CFS
- จำลองเหตุการณ์
- ประสิทธิภาพของ CFS