

# Scheduling



© 2022 מערך הפעלה

## Operating Systems

### Lecture 3 – Scheduling

Dr. Marina Kogan-Sadetsky

זמן תהליכי:

אחד האלגוריתמים החשובים במערכת ההפעלה.

הweeneyון באלגוריתמים האלו הוא שהאלגוריתם צריך לנחש נכון התהיליך הבא שהוא צריך להריץ.

## Course Syllabus

1. Introduction
2. Process Management
3. Scheduling algorithms
  - Scheduling criteria
  - Scheduling algorithms
  - Unix scheduling
  - Linux scheduling
  - Windows scheduling
4. Synchronization
5. Memory Management
6. File Systems
7. Virtualization

נגיד קритריונים לאיכות אלגוריתמים.

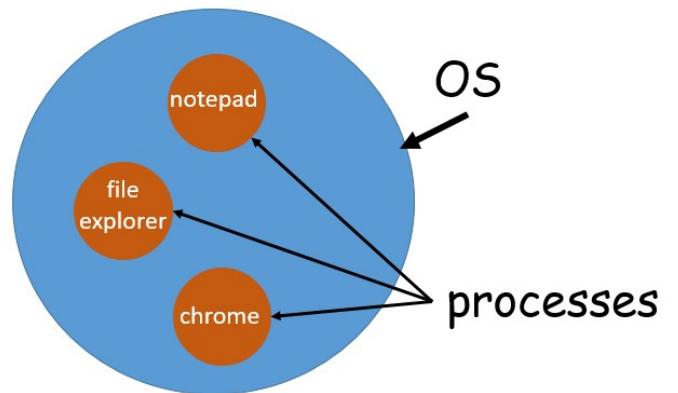
יותר מדויק או פחות מדויק, מה יותר זמן ריצה ומה פחות זמן ריצה.

## Scheduling: outline

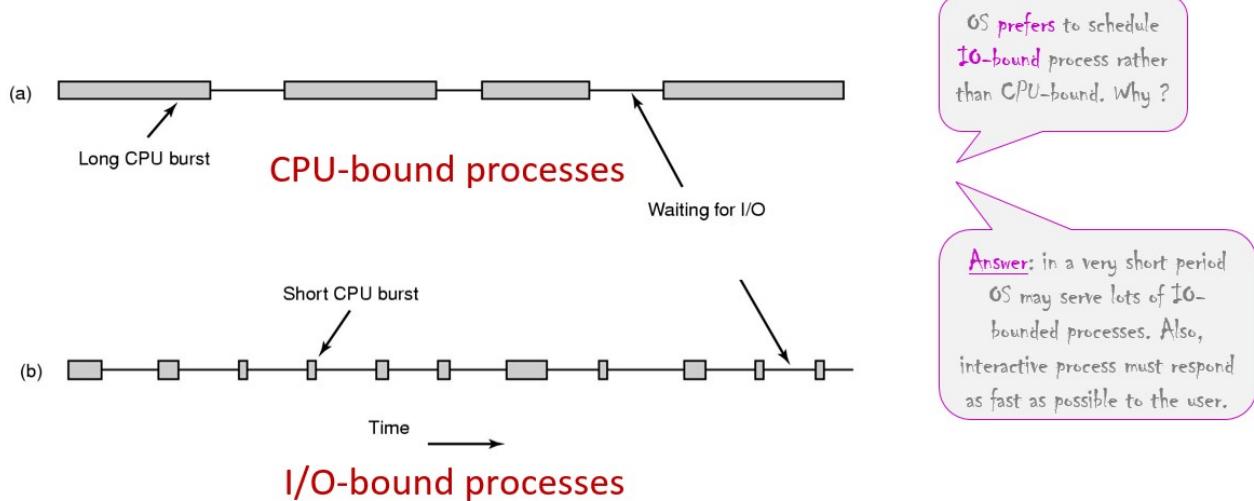
- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux scheduling
- Windows scheduling

## Scheduling: high-level goals

Who will run ?  
When will it run ?  
For how long ?



# Processes – types of behavior



יש 4 קритריונים שצורך ללמידה למבחן.

הקריטריון המרכזי הוא ההתנהגות של התהילה. במהלך פעילותו, מערכת הפעלה לומדת את ההתנהגות של התהילה. נקודת ההתחלה של מערכת הפעלה היא שהיא אינה מכירה את התהילה.

לדוגמא, אם אני מקצה זמן מעבד לתהילה והתהילה מוחזיר אותו בשל פעולות IO, הזמן שעובר מהרגע שהטהילה קיבל את הזמן ועד שהוחזיר אותו, הוא זמן אrror.

נניח שיש לנו תהילה שבכל פעם שהוא מרים, הוא משתמש בפחות זמן מהזמן שהוקצה לו, כלומר, הוא מבצע בעיקר פעולות IO.

תהליכים אלו, שתלויים ב-IO, הם אלו שמערכת הפעלה תעדיף לתת להם זמן מעבד. הסיבה לכך היא שהם מתקשרים עם המשתמש. לדוגמה, כמשמעות, יותר חשוב לי שהאפליקציה של WhatsApp קיבל זמן מעבד.

בנוסף, ישנו תהלים שהם יותר חישוביים, כלומר, הם מבצעים יותר אינטראקציות עם ה-CPU. מערכת הפעלה מודדת את הזמן שהטהילה מבצע באופן קל ומדויק את התעדוף שלו בהתאם.

# Scheduling algorithms: quality criteria

- Fairness:** *comparable processes should get comparable service*
- Efficiency:** *maximize CPU utilization and I/O devices utilization*
- Response time:** *minimize the time between command submission and generation of first output*
- Waiting time:** *minimize the time processes wait in Ready Queue*
- Turnaround time:** *minimize (average) time between process start and termination (for jobs)*
- Throughput:** *maximize number of completed processes per time unit (for jobs)*
- Deadlines / Predictability :** *when command must be completed in some predefined time (for real-time applications)*

**CPU utilization**  
- % of time CPU is not idle

Note that some criteria are conflicting:

- By giving CPU to interactive processes each time they are READY, we raise Response time, but harm Turnaround time.
- For higher Throughput we should prefer shorter processes, but we harm Waiting time - long processes might be starved.

**מددים:**

ישנם 4 מددים שאינם חשובים (אלו שבוחר) וישנם 3 מהם מאוד חשובים.

א. הוגנות: מערכת הפעלה משתמשת להיות הוגנת במובן שהיא מתייחסת לתהליכים שהיא רואה כשקלים מנוקודת מבט המשתמש באופן דומה.

ב. יעילות: המטרה היא למקסם את הניצול של המעבד, אחד מהמשאים החשובים ביותר. אם מעבד מנצל באופן תות-אופטימלי, זה מצביע על אלגוריתם זמן גרוע.

ג. זמן תגובה: זה הזמן שעובד מראה מקבלת עד שהפלט הראשון מיוצר. מدد זה חשוב מנוקודת מבט המשתמש. לדוגמה, אם אמ' תוכנית לוקחת הרבה זמן לחשב פלט, רוב האפליקציות יצאו בר התקדמות או מחווין אחר למשתמש. אנחנו מתרבונים בזאת מנוקודת מבט של מערכת הפעלה ומתקודת מבט של המשתמש.

ד. זמן המתנה: תהליך יכול להמתין לנוכח בתור אם מערכת הפעלה תמיד מעדיפה תהליכים אחרים על פניו. בעוד שלמערכת הפעלה אין משנה רעה, משתמש כן. המשתמש רוצה שהתהליכים שלו ירצו. מערכת הפעלה מודעת לכך. לדוגמה, בلينוקס, כל 20 יחידות זמן, היא נותנת לכל תהליך קצר זמן ריצה.

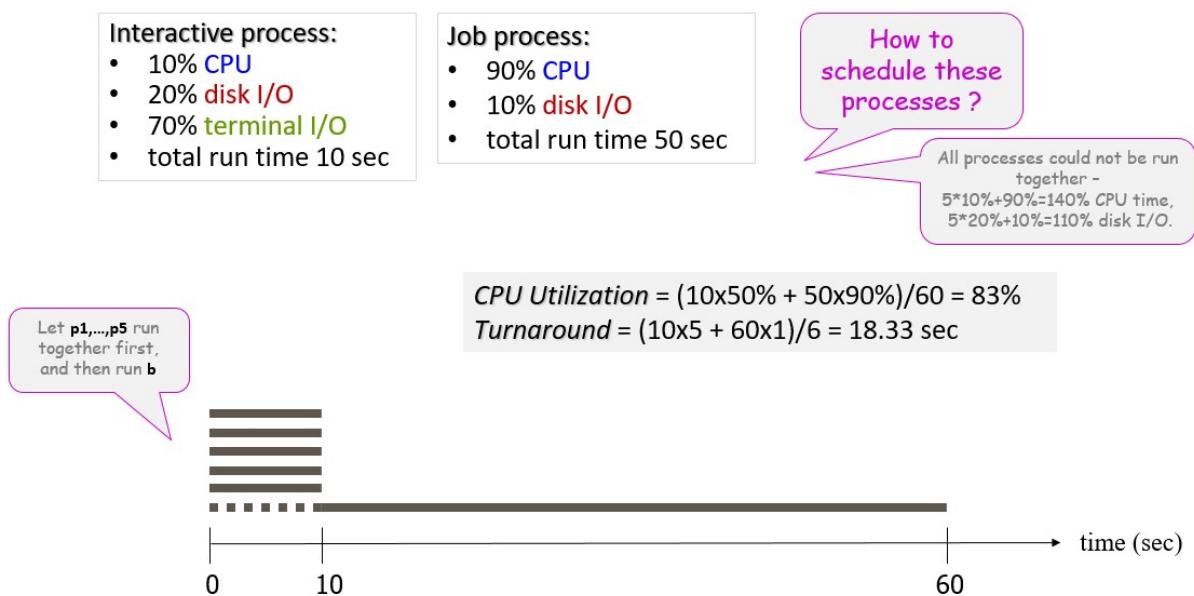
ה. זמן סיבוב: זה הזמן הכלול שנדרש מהתחלת עד סיום של תהליך. אנחנו מחשבים את הממוצע של זה במספר תהליכים. ממד זה בערך חסר משמעות מאחר ונחנו משאים אפליקציית פתוחות למשך תקופות ממושכות. הוא חשוב רק לתהליכים חשובים.

6.-DDLINIM / ניבוי: זה חשוב למשתמש בזמן אמיתי להם דಡליינים. לדוגמה, סיבוב של הגה ברכב. אם תהליך מחמיא את הדדלין שלו, זה יכול להוביל לתוצאות רציניות.

כל מהמדדים האלה משחק תפקיד באופן שבו מערכת הפעלה מזמן וניהלת תהליכים. מערכת הפעלה חייבת לאזן בין הממדדים כדי להבטיח ניהול ניהולiesel של משאבי המערכת.

# CPU Utilization vs. Turnaround Time Example

We have single CPU, 5 interactive processes **p1,...,p5**, and one job process **b**.



שאלה:

נניח שיש לנו מעבד אחד, חמשה תהליכי אינטראקטיביים ותהליך עבודה אחד. תהליך אינטראקטיבי הוא תהליך שביצוע רוב הזמן קריאות ISO לדיסק ולמסוף, ומנצל רק 10% מהמעבד. תהליך עבודה הוא תהליך שמשתמש בעיקר במעבד ובמצע קריאות ISO לדיסק רק 10% מהזמן. בנוסף, נתון זמן הריצה של כל תהליך אינטראקטיבי הוא 10 שניות, וזמן הריצה של תהליך העבודה הוא 50 שניות. איזה זמן יהיה ה"טוב ביותר"?

תשובה:

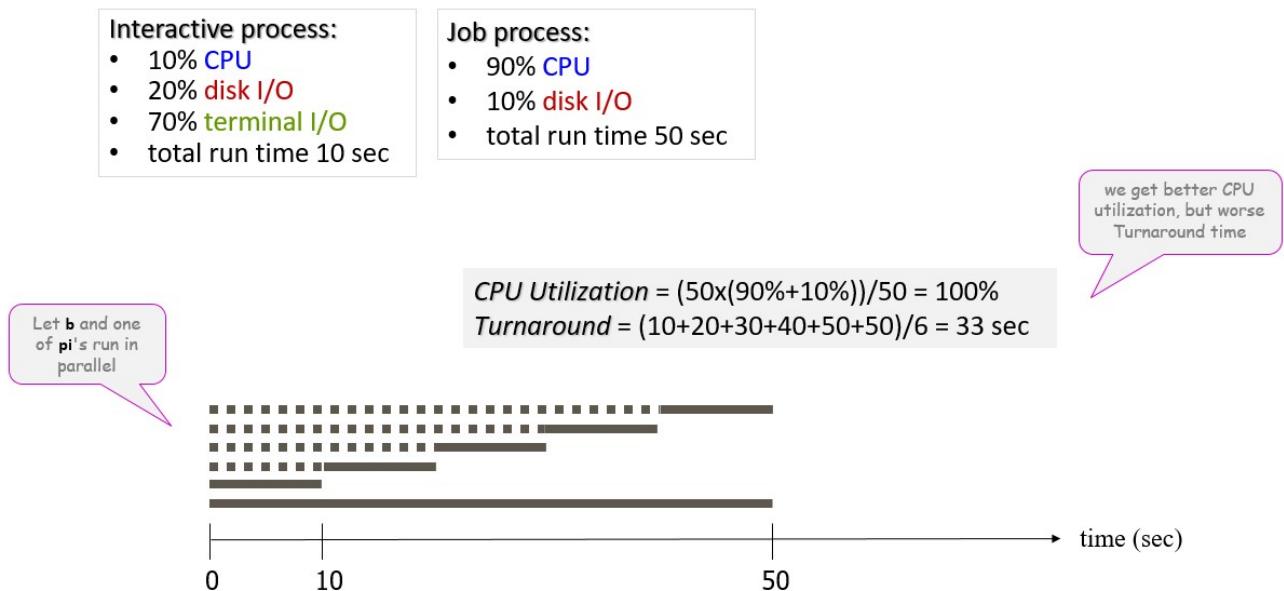
כדי להבין מהו ה"טוב ביותר", אנחנו צריכים להתחיל לחשב על ממדים לבחירת אלגוריתם. אילו ממדים חשובים לנו? יש לנו תהליכי אינטראקטיביים שונים מהמשתמש רוב הזמן, ותהליך עבודה שמנצל את המעבד באופן משמעותי. כאן, ממד ה-Turnaround והיעילות הם החשובים ביותר.

רעיון ראשון הוא לחתה לתהליכי האינטראקטיביים לרווח קודם, ואז להריץ את תהליך העבודה. כאשר אנחנו מחשבים את הממדים, אנחנו מגלים שהnicol של המעבד הוא 83%, וה-Turnaround הממוצע הוא 18.3 שניות. אף על פי שה-Turnaround נמוך, אנחנו לא מנצלים את המעבד למלאו.

אז, אולי ננסה גישה אחרת שתנצל את המעבד באופן מלא.

## CPU Utilization vs. Turnaround Time Example

We have single CPU, 5 interactive processes **p1,...,p5**, and one job process **b**.



### רעיון שני:

נתחיל עם ריצת ה-JOB, ובמהלך כל שלב נבצע החלפת הקשרים עם התהיליכים האינטראקטיביים. התכנית האינטראקטיבית לא באמת זקוקה לכל הכוח של המעבד, אלא רק להוראות קלט/פלט.

### ニיצול מעבד:

במקרה זה, יש לנו ניצול של 90% במשך 50 שניות, וניצול של 10% במשך 50 שניות, מכיוון שככל פעם שתהיליך אינטראקטיבי מסתיים, מערכת הפעלה בוחרת תהיליך אינטראקטיבי אחר. כך, ניצול המעבד הגיע ל-100%, מה שהוא מצוין.

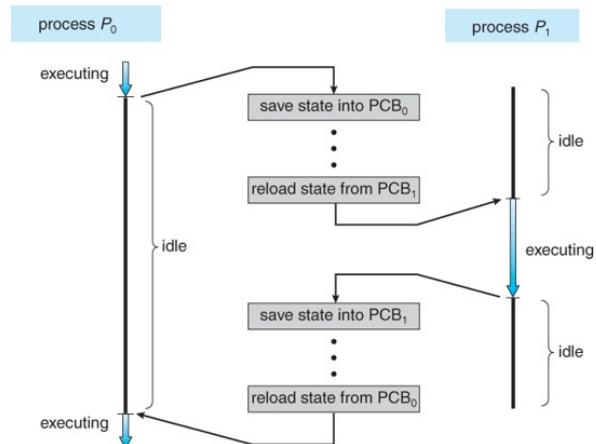
### Turnaround:

ה-JOB רץ מהרגע הראשון, ולכן זמנו הוא 50 שניות. אם התהיליך האינטראקטיבי הראשון שהמערכת בחרה להריץ לא ממתין, זמנו הוא 10 שניות. התהיליך האינטראקטיבי השני ממתין 10 שניות ואז רץ 10 שניות, וכך הלאה. turnaround שלו הוא 20 שניות. באותו אופן, התהיליך האינטראקטיבי השלישי מקבל 30, הרביעי 40, וה חמישי 50. בסה"כ, זמן ה-Turnaround הממוצע הוא 33 שניות.

במקרה זה, שיפרנו את מדריך ניצול המעבד אך פגנו במדד זמן ה-Turnaround. עם זאת, זה עדין מצוין, לאחר ואנחנו מדיפים את המדריכים באדום על פני המדריכים בשחור, כפי שמוצג בשקף השישי.

# Context switch overhead - example

- Assume a context switch takes 5 milli
- Assume 10 running processes
- Switching every **20 milli** wastes **20%** of CPU time
  - ✓ for each 20 milli run-time, we waste 5 milli for switching, which means wasting of 20%
- Switching every **500 milli** might lead to  $500 \text{ milli} * 10 = 5 \text{ second}$  response time
- possible solution:** settle switch time of **100 milli**



נקודה חשובה לשיקול היא החסרון של החלפת הקשרים. נניח שהחלפת הקשרים לוקחת 5 מילি-שניות, ויש לנו 10 תהליכיים. אם אנחנו מבצעים החלפה כל 20 מילি-שניות, אנחנו מזבזים 20% זמן ה-CPU. אם אנחנו מבצעים החלפה כל 500 מילি-שניות, זמן התגובה שלנו מושפע. פתרון אחד שאנו יכולים לשקל הוא לבצע החלפת הקשרים באינטרוולים של 100 מילি-שניות.

נקודות חשובות נוספות לשיקול:

א. אם המעבד בחר לבצע החלפת הקשר עם אותו תהליך שהוא כבר הרץ, זה לא בהכרח אומר שהוא פשוט ימיר להרץ אותו. המעבד מגיב למצבים שונים ומקבל החלטות בהתאם. למשל, יתכן שהמעבד יבצע איבוי, בין אם יש צורך בהחלפת הקשר ובין אם אין.

ב. במעבדים של 64 ביט, מערכת הפעלה משתמשת חלק מתמונת הזיכרון של כל תהליך. זו אחת הסיבות שהחלפת תמונה זיכרון אינה פעולה קרה מדי.

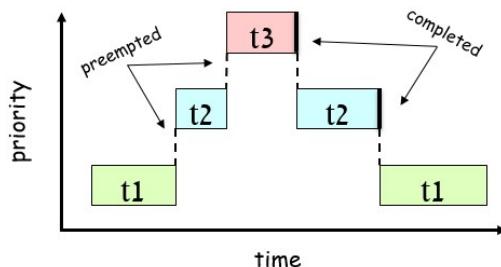
# Preemptive vs. non-preemptive scheduler

## ❑ Preemptive scheduler

process may be rescheduled to run latter even if its CPU burst time is not finished

## ❑ Non-preemptive scheduler

process is RUNNING until it voluntarily releases CPU (BLOCK, terminates, or yield(s))



אלגוריתמי תזמון מתחלקיים לשני סוגים עיקריים:

א. **תזמון מקדים (Preemptive Scheduler)**: באלאgorיתם זה, המזמן בוחר תחיליך ומאפשר לו לרצץ למשך זמן מוגבל. אם התהיליך ממשיך לרצץ בסוף פרק הזמן שהוקצב לו, התהיליך מועבר למצב השהייה והזמן בוחר תחיליך אחר לקבלת זמן הריצה. כאן, יש לנקח בחשבון פסיקות שעון בסוף כל אינטראול זמן, כדי להחזיר את השליטה על המעבד למזמן.

ב. **תזמון לא-מקדים (Non-Preemptive Scheduler)**: באלאgorיתם זה, המזמן בוחר תחיליך לקבלת זמן ריצה והתהיליך ממשיך לרצץ עד שהוא מעביר את עצמו למצב Block או שהטהיליך משחרר את המעבד באופן יוזם. במצב זה, אין אפשרות להשווות את פעולת התהיליך, אפילו אם הוא רץ למשך שניות.

## When are Scheduling Decisions made ?

1. Upon *clock interrupt*
2. Process switches from *RUNNING* to *BLOCKED*
3. New *process created*
4. Process switches from *BLOCKED* to *READY*
5. Process *terminates*
6. Process *yield()*s

Which of above possible for non-preemptive scheduling?

2, 5, 6

החלטות תזמון מתתקבלות במקרים הבאים:

- א. כאשר מתרחשת פסיקת שעון.
- ב. כאשר תהליך משנה מצב מ-Running ל-Blocked.
- ג. כאשר נוצר תהליך חדש.
- ד. כאשר תהליך משנה מצב מ-Blocked ל-Ready.
- ה. כאשר תהליך מסיים את ריצתו.
- ו. כאשר תהליך מוותר על זמן הריצה שלו.

באלגוריתמי תזמון לא-מקדים, הנקודות היחידות שבهن ניתן לבצע תזמון הן ב', ה' וו'.

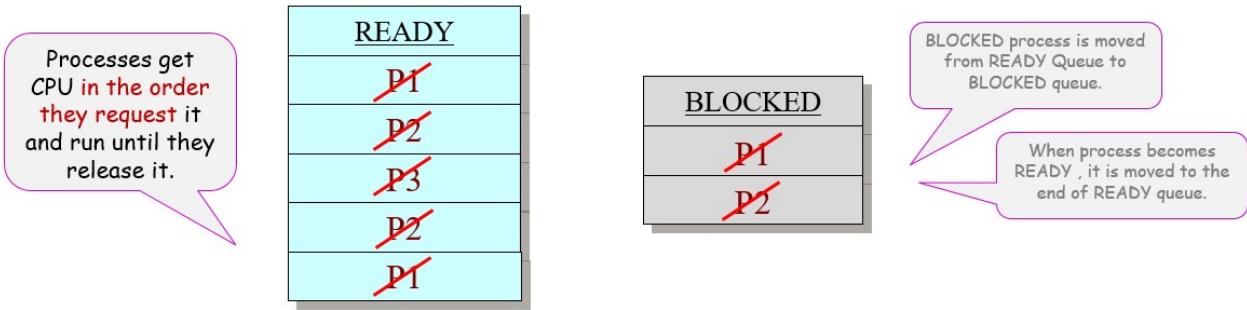
## Scheduling: outline

- Scheduling criteria
  - Scheduling algorithms
  - Unix scheduling
  - Linux scheduling
  - Windows scheduling

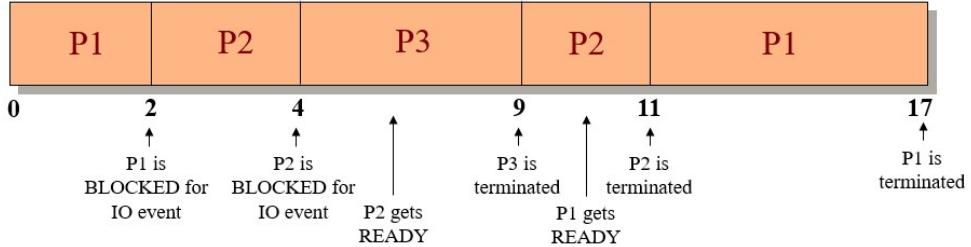
## אלגוריתמי תזמון

הערה: בדוגמאות יש שימוש רק במעבד עם ליבת אחת רק לשם הפשטות. כאן נגיד יש שימוש בثور, אז אם יש כמה ליבות אז תהיה חלוקה לכמה תורים. מה שמשמעותו בחלק זהה זאת ההחלטה עצמה - העיקרון.

# Non-preemptive First-Come-First-Served (FCFS)



If they arrive in the order P1, P2, P3, we get:



## אלגוריתם התזמון "הראשון שmagיע, הראשו שמתבצע" (First-Come-First-Served, FCFS)

הוא אלגוריתם לא-מקדים ופשוט להבנה. האלגוריתם מבצע את התהליכים לפי סדר הגעתם לתור הממתין. ברגע שההילך מגיע למעבד, הוא מתחילה לרווח מיד ומקבל את כל זמן הריצה שהוא דרוש. תהליכים שmagיעים לאחריו מתווספים לסוף התור וממתינים לתורם עד שההיליך הנוכחי מסיים את ריצתו או מתבצע החלפת הקשר.

### הוראות לביצוע האלגוריתם:

- כארה תהליך מבקש זמן ריצה מהמעבד, הוסיף אותו לסוף התור.
- אם המעבד פנוי, היזה את התהליך הראשון בתור למעבד.
- הרץ את התהליך במעבד עד שהוא מסיים את ריצתו או עובר למצב Block.
- כארה התהליך מסיים את ריצתו או מתבצע החלפת הקשר, היזה את התהליך הבא מראש התור למעבד.
- חזר לשלב 3.

### יתרונות:

- קל להבנה ולתכנות.
- הווגן - התהליך שהמתinan כי הרבה זמן מקבל את זמן הריצה.
- סיבוכיות זיכרון נמוכה - נדרש רק רישימה מקושרת לניהול התור.
- סיבוכיות זמן ריצה נמוכה - נדרש רק פעולות הכנסה והוצאה מהתור.

### חסרונות:

אלגוריתם זה עשוי להיות לא אפקטיבי במקרים שבהם התהליך CPU-Bound מתחילה לרווח ומעכב את הריצה של תהליכי I/O. במקרה כזה, אלגוריתם זמן מקדים שימושה את התהליך החישובי באינטראולים קצרים יכול להיות יעיל יותר.

במקרים מסוימים, יתכן שאלגוריתם תזמון מוקדים יהיה יותר, כיוון שהוא מאפשר לתחליכים קצרים לרווח ולהסתיים מהר יותר, מה שמקטין את זמן ההמתנה של תחאליכים ארוכים יותר. הדבר בא לידי ביטוי בדוגמה הבאה:

## Non-preemptive First-Come-First-Served (FCFS)

| Process | Burst time (milli) |
|---------|--------------------|
| P1      | 24                 |
| P2      | 3                  |
| P3      | 3                  |

Let's take  
smaller  
example.

If they arrive in the order P1, P2, P3, we get:



Average waiting time:  $(0+24+27) / 3 = 17$

**burst time:**

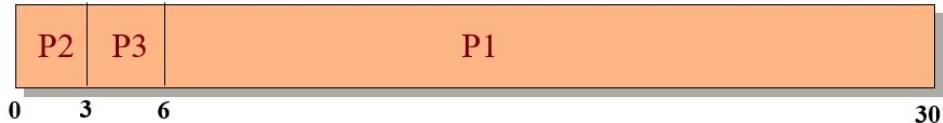
פרק הזמן עד שתהיליך נכנס ל-block.

כאן התהיליך הראשון מקבל זמן ריצה, רץ במשך 24 מילישניות ואז השניים האחרים מקבלים זמן ריצה. זה לא עיל. כאן התהיליך הראשון שהוא אם זה שדורש הכיר הרבה זמן ריצה אומנם מס'ם מוקדם אבל תהיליך 2 צריך להמתין 24 מילישניות והשלישי צריך להמתין 27 מילישניות.

## Non-preemptive First-Come-First-Served (FCFS)

| Process | Burst time (milli) |
|---------|--------------------|
| P2      | 3                  |
| P3      | 3                  |
| P1      | 24                 |

What if they arrive in the order P2, P3, P1?



Average waiting time:  $(0+3+6) / 3 = 3$

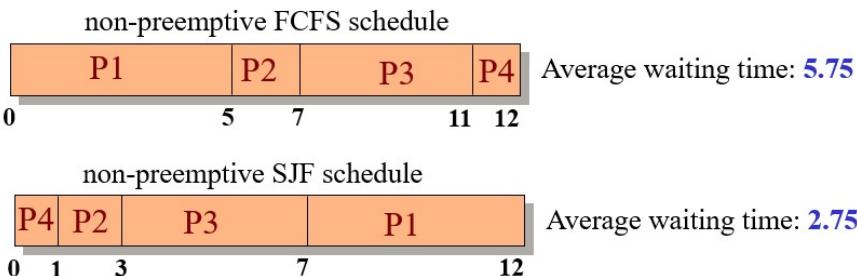
It seems like it is  
better to run  
shortest jobs first.

במקום זה ניתן היה לתת את זמן הריצה לתחאליכים שדורשים הכי פחות זמן, ובכך לkürק את זמן ההמתנה לתחאליך הארוך.

# Non-preemptive Shortest Job First (SJF)

| Process | Burst time (milli) |
|---------|--------------------|
| P1      | 5                  |
| P2      | 2                  |
| P3      | 4                  |
| P4      | 1                  |

CPU is assigned to the process that has **shortest CPU burst**. This value is pre-defined or is approximated.



# Non-preemptive Shortest Job First (SJF)

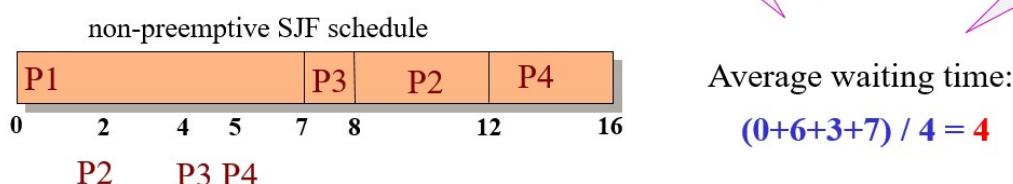
| Process | Arrival time | Burst time |
|---------|--------------|------------|
| P1      | 0            | 7          |
| P2      | 2            | 4          |
| P3      | 4            | 1          |
| P4      | 5            | 4          |

Note that the processes arrive at different times.

How can we do better?

Preemptive scheduler!

What do we need for this?



clock produces hardware interrupt, which causes CPU to run OS Scheduler

## Non-preemptive Shortest Job First (SJF)

אלגוריתם ה-SJF הוא אלגוריתם של תזמון באצווה שמניח שזמן הריצה של התהליכים ידועים מראש. המתזמן בוחר את העבודה הקצרה ביותר תחילת, מה שוביל לשיפורים משמעותיים בזמן המתנה הממוצע. עם זאת, אלגוריתם זה הוא אופטימלי רק כאשר כל העבודות זמינים בו זמן נייד.

תיאור האלגוריתם שלב אחר שלב:

1. הגעת העבודה: האלגוריתם מניח שכל העבודות זמינים בהתחלה. חשוב לציין שאלגוריתם זה הוא אופטימלי רק כאשר כל העבודות זמינים בו זמן נייד. אם העבודות מגיעות בזמןים שונים, האלגוריתם עשוי לא לספק את זמן המתנה הממוצע הקצר ביותר.

**בחירת העבודה:** המתזמן בוחר את העבודה עם זמן הביצוע הקצר ביותר תחילת. זמן הביצוע מתייחס לזמן 2. שהתחילה זוקק לו כדי להשלים את הביצוע שלו. אם לשני תהליכיים יש את אותו זמן הביצוע, המתזמן יכול לבחור אחד מהם.

**ביצוע העבודה:** העבודה שנבחרה מבוצעת באופן שאין מקדים, כלומר היא רצה עד הסיום ללא הפסקה על ידי המתזמן. 3.

**בחירה העבודה הבאה:** ברגע שהעבודה הנוכחית מסיימת את הביצוע, המתזמן בוחר את העבודה הקצרה ביותר מהעבודות הנותרות. התהילה זהה ממשיך עד שכל העבודות מבוצעות. 4.

**חישוב זמן המתנה הממוצע:** זמן המתנה הממוצע מחושב על ידי חיבור זמני המתנה של כל ה手続きים 5. וחילוקם במספר ה手続きים. אלגוריתם SJF מכוון לסייע את זמן המתנה הממוצע.

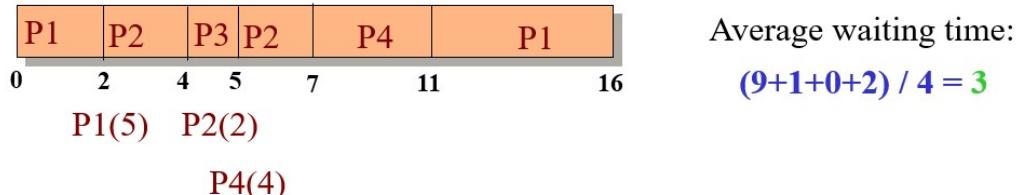
חשיבותו של אלגוריתם SJF יש מגבלה משמעותית: הוא דורש מידע מראש על זמן הביצוע של כל העבודות, מה שלא תמיד אפשרי במצבות. בנוסף, הוא יכול להוביל להタルמות מתקלבים ארוכים יותר אם手続きים קצרים ממשיכים להגיע. 6.

בشكل הבא, אנו נלמד גם על ארסה מקדימה של אלגוריתם SJF בשם SRTN (Shortest Remaining Time Next) אלגוריתם זה תמיד בוחר את התהילה עם זמן הריצה הנותר הקצר ביותר. כאשר עבודה חדשה מגיעה, זמן הביצוע הכלול שלה משווה לזמן הנותר של התהילה הנוכחי. אם העבודה החדשה זוקקה לפחות זמן להשלמה מאשר התהילה הנוכחי, התהילה הנוכחי מושהה, והעבודה החדשה מתחילה. תכנית זו מאפשרת עבודות חדשות קצרות לקבל שירות טוב.

# Preemptive SRTF – (Shortest Remaining Time First)

| Process | Arrival time | Burst time |
|---------|--------------|------------|
| P1      | 0            | 7          |
| P2      | 2            | 4          |
| P3      | 4            | 1          |
| P4      | 5            | 4          |

SRTF - preemptive SJF schedule

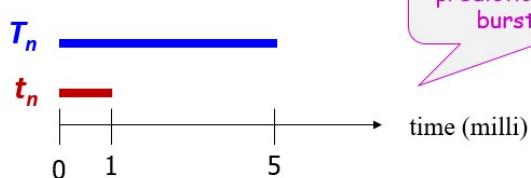


## Approximating Next CPU-burst Duration

- $T_n$  = ***predicted*** value for the current CPU burst
- $t_n$  = ***actual*** length of  $n^{th}$  CPU burst
- $T_{n+1}$  = ***predicted*** value for the next CPU burst
- for  $0 \leq \alpha \leq 1$  define exponential average:

$$T_{n+1} = \alpha t_n + (1 - \alpha) T_n$$

larger  $\alpha$  means that we believe that next CPU burst would be similar to the previous one



What should be our prediction for next CPU burst duration?

## Shortest Remaining Time First (Preemptive SRTF)

אלגוריתם ה-Shortest Remaining Time First (SRTF), המכונה גם אלגוריתם ה-Job First, הוא אלגוריתם לתזמון תהליכי שמעניק עדיפות לתהליכים על פי זמן הביצוע הנותר שלהם. ההנחה הבסיסית כאן היא שזמן הביצוע הכולל של כל תהליך מוכך מראש.

תיאור האלגוריתם שלב אחר שלב:

א. **הגעת תחילך:** כאשר תחילך חדש מגיע, זמן הביצוע הכולל שלו משווה בזמן הביצוע הנותר של התחלך שمرיאץ כרגע.

ב. **בחירה תהיליך:** המתזמן תמיד בוחר את התהיליך שזמן הביצוע הנותר שלו הוא הכי נמוך. אם התהיליך החדש דורש פחות זמן להשלמה מאשר הזמן הנותר של התהיליך הנוכחי שMRIIZ כרגע, התהיליך הנוכחי מושעה, וההתהיליך החדש מתחליל.

ג. **החלפת הקשרים:** אלגוריתם זה מאפשר הפסקה, ככלומר, תהיליך שMRIIZ יכול להישגה כדי לאפשר לתהיליך חדש עם זמן נותר קצר יותר להתבצע. תוכנה זו מבטיחה שימושות קצרות לא נשארות מחכות למשימות ארוכות להשללים, מה שמשפר את הייעילות הכלכלית של המערכת.

ד. **ערכת זמן הביצוע הבא:** האלגוריתם משתמש בטכנית בשם "הזדקנות" כדי לחזות את זמן הביצוע הבא על פי הערכים הקודמים. נניח שהזמן המוערך לתהיליך מסויים הוא  $T_0$ , והריצה הבאה שלו מודדת להיות  $T_1$ . ההערכה החדשה מתקדמת על ידי לקיחת סכום משוקל של שני המספרים האלה, ככלומר,  $T_1 = a \cdot T_0 + (1 - a) \cdot T_1'$ . המקבם 'a' קובע כמה מהר התהיליך של ההערכה שוכח את הריצות הישנות. אם  $a = 1/2$ , ההערכה החדשה הופכת להיות  $T_1 = T_0/2 + T_1'/2$ .

ה. **זמן קוונטום:** הזמן האמיתי,  $t_q$ , מוגבל על ידי ה-Quantum Time. Quantum Time הוא הזמן המרבי שתהיליך יכול להחזיק את ה-CPU ללא שחרור.

ו. **עדכון תחזית:**  $t_q + T_0$  הוא עדכון של התחזית על פי  $t_q$ . העדכון מתבצע על ידי שימוש במקדם אלפא ( $\alpha$ ), שהוא שבר. אם  $\alpha$  קרובה לאפס, השינויים בתחזיות יהיו איטיים יותר. אם  $\alpha$  קרובה לאחת, השינויים בתחזיות יהיו מיידיים יותר.

יש לציין כי הייעילות של אלגוריתם SRTF תלויות במידה רבה בבדיקה של ההערכות של הזמן. אם הם לא מדויקים, האלגוריתם עשוי לא לעבוד כפי שצפוי. כמו כן, כמו כל האלגוריתמים המקדמים, SRTF יכול לסבול מהעומס של החלפת תהליכיים תדירה.

## Round Robin (RR)

- ❑ Each process gets a small unit of CPU time (**time-quantum**), usually 10-100 milliseconds
- ❑ For  $n$  ready processes and time-quantum  $q$ , maximum waiting time is  $(n - 1)q$
  
- ❑ If time-quantum  $\sim$  context switch time (or smaller)
  - ➔ faster response
  - ➔ reduce CPU utilization (CPU time is wasted on switches)
  
- ❑ If time-quantum  $>>$  context switch time
  - ➔ better CPU utilization
  - ➔ slower response



cutet ניבור לאלגוריתמי זמן במערכות אינטראקטיביות כמו במחשבים אישיים, שרתים וכו'.

### Round Robin

אלגוריתם התזמון Round Robin הוא אלגוריתם של חלוקת זמן, הנמצא בשימוש נרחב בשל פשוטותו והונגוותו. זהה אחד מהאלגוריתמים הותיקים והנפוצים ביותר שימושים במערכות אינטראקטיביות כמו מחשבים אישיים ושרתים. כך הוא עובד:

א. **הגעת תחיליך:** כאשר תחיליך מגיע, הוא מתווסף לרשימה של תהליכי הניתנים להרצה.

ב. **הקצת קוונטים:** לכל תחיליך מוקצה מרוחץ זמן קבוע, המכונה קוונטום, במהלךו הוא מושך לרווח.

ג. **בחירה תחיליך:** המתזמן בוחר את התחליך הראשון מהרשימה ומאפשר לו לרווח למשך הקוונטים שהוקצה לו.

ד. **החלפת הקשר:** אם התחליך עדיין רץ בסוף הקוונטים שלו, ה-*CPU* מושחה, והתחליך מועבר לסוף הרשימה. ה-*CPU* אז מועבר לתחליך הבא ברשימה. אם התחליך מסיים או מונע לפני שהקוונטים שלו נגמר, ה-*CPU* מוחלף לתחליך הבא ברשימה.

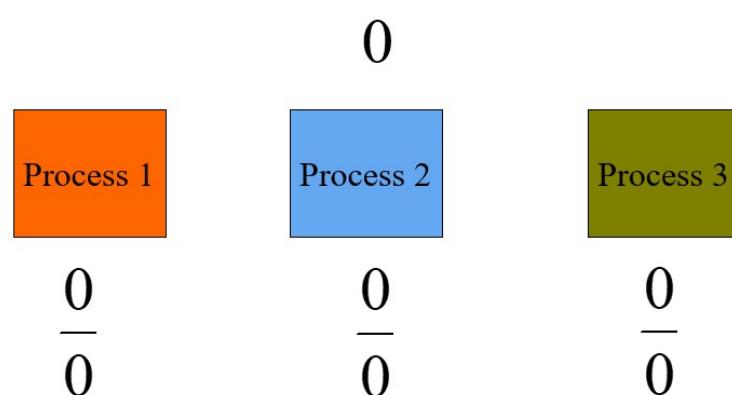
ה. **חזרה על המבחן:** מכיון זה ממשיר, כאשר המתזמן עבר על הרשימה של תהליכי הניתנים להרצה, דבר המאפשר לכל תחליך לרווח למשך הקוונטים שהוקצה לו.

האתגר העיקרי עם אלגוריתם Round Robin הוא לקבוע את האורך המתאים לקוונטים. אםekoונטים קצר מדי, ה-*CPU* עשוי לבודד זמן משמעותי מזמן שימושו בינם לבין תהליכי, מה שיכל להוביל לחוסר יעילות. אםekoונטים ארוך מדי, זמן התגובה עשוי להיות ארווע, במיוחד במערכות עם הרבה תהליכי.

אלגוריתם Round Robin פשוט למימוש ומספק חלוקה הוגנת של זמן *CPU* בין תהליכי. עם זאת, הוא יכול לסבול מעליות החלפת הקשר אבסולוטית אםekoונטים אינם מוגדר באופן מתאים. חשוב גם לציין שכל תהליכי מחשבים שווים באlgorigthm זה; אין עדיפות שנייה לתחליך מסוים.

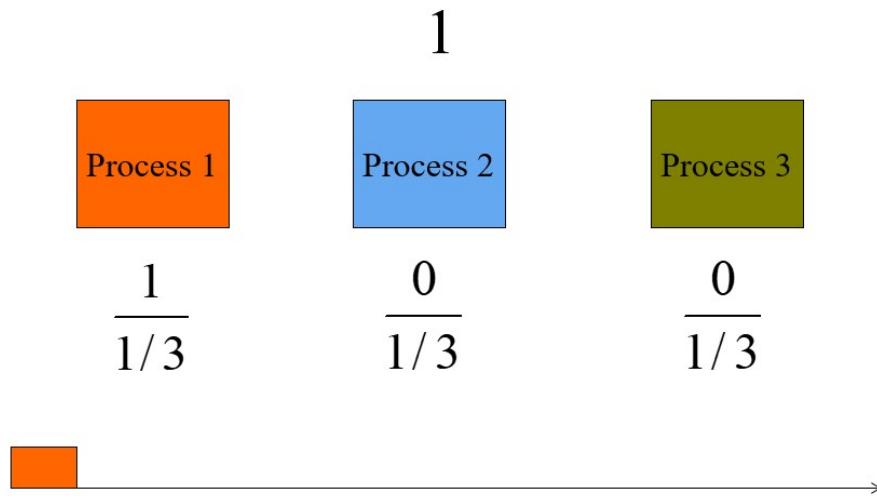
## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time



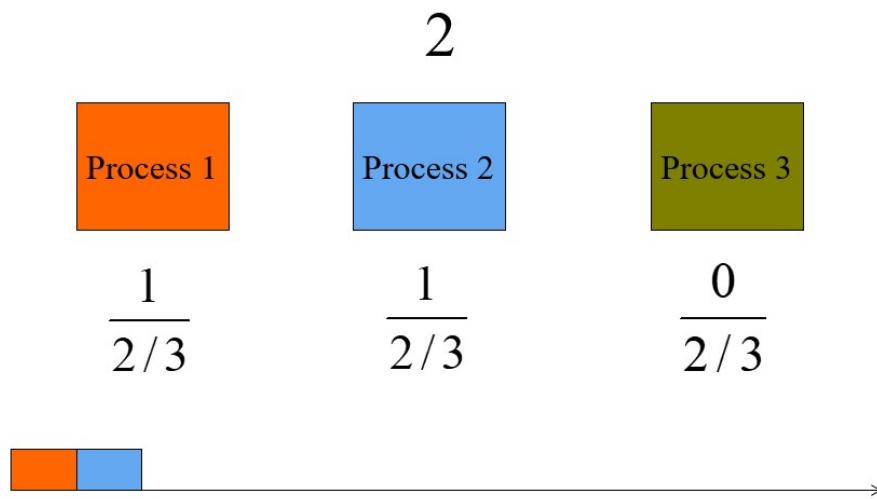
## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time



## Guaranteed (Fair-share) Scheduler

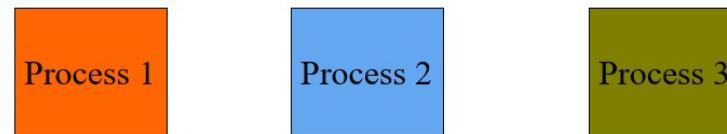
given  $n$  processes, each process should get  $1/n$  of CPU time



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

3



$$\frac{1}{3/3}$$

$$\frac{1}{3/3}$$

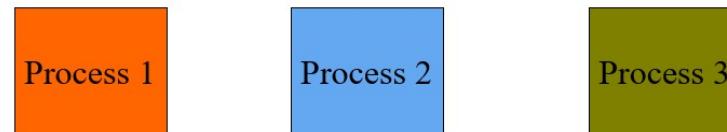
$$\frac{1}{3/3}$$



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

4



$$\frac{2}{4/3}$$

$$\frac{1}{4/3}$$

$$\frac{1}{4/3}$$



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

5



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

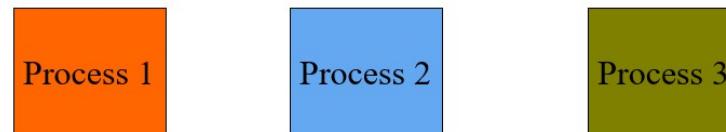
6



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

7



$$\frac{3}{7/3}$$

$$\frac{2}{7/3}$$

$$\frac{2}{7/3}$$



## Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time

8



$$\frac{3}{8/3}$$

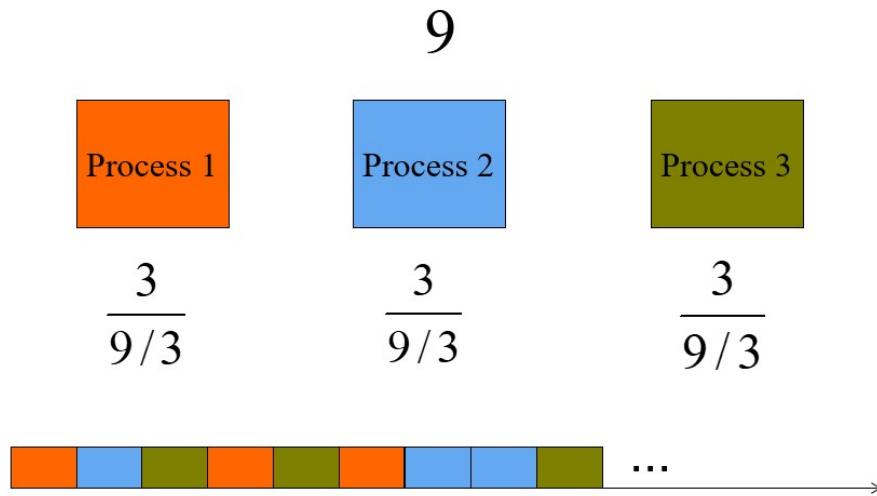
$$\frac{3}{8/3}$$

$$\frac{2}{8/3}$$



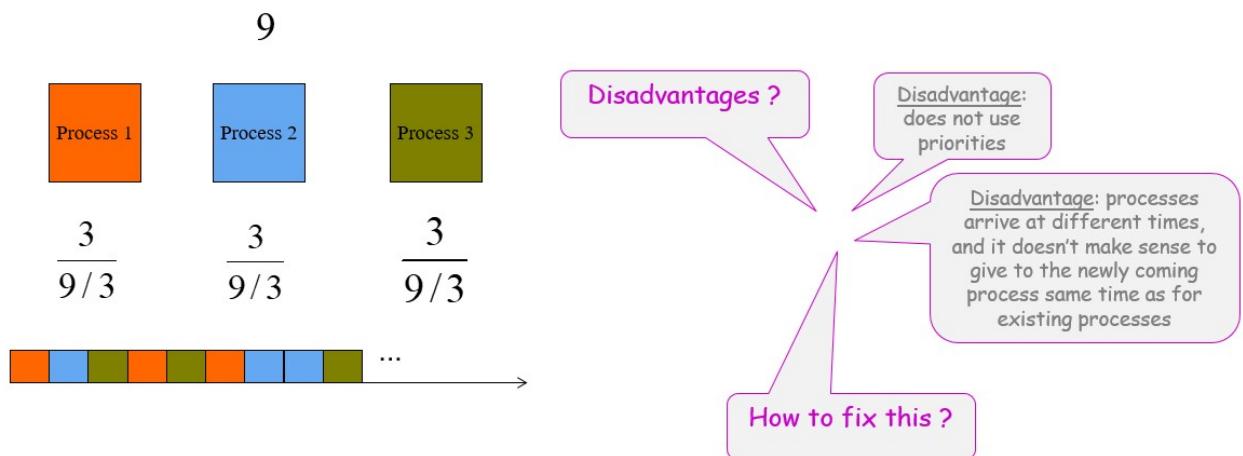
# Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time



# Guaranteed (Fair-share) Scheduler

given  $n$  processes, each process should get  $1/n$  of CPU time



## Guaranteed (Fair-share) Scheduler

הערות:

יש דברים מיוחדים בשקופיות: המכנה מיותר, דברים מיוחדים במדידה.

אלגוריתם הזמן "המתזמן המובטח (חלוקת הוגנת)" מספק הבטחות אמיתיות של ביצועים למשתמשים.  
אלגוריתם זה מבטיח שם יש 'ח' תהליכי, כל תהליך יקבל  $1/n$  זמן ה-CPU. הנה איך זה עובד:

א. הגעת תהליך ומעקב אחר שימוש ב-CPU: כאשר תהליך מגיע, המערכת שומרת על מעקב אחר כמה CPU כל תהליך השתמש מאז שנוצר.

**ב. חישוב זכאות בזמן CPU:** המערכת אז מחשבת את כמות זמן ה-CPU שכל תהליך זכאי לה. זה מחושב כזמן מאז שההתהליך נוצר מחולק על מספר התהליכיים (n).

**ג. חישוב יחס זמן CPU בפועל בזמן CPU שהוקצב:** המערכת גם יודעת את זמן ה-CPU הבפועל בכל תהליך צרך. היא מחשבת את היחס של זמן ה-CPU הבפועל שנוצר לזמן ה-CPU שההתהליך זכאי לו. יחס של 0.5 אומר שההתהליך היה רק חצי ממה שהוא אמור לקבל, ויחס של 2.0 אומר שההתהליך היה פי שניים ממה שהוא זכאי לו.

**ד. ביצוע התהליך:** לאחר מכן, האלגוריתם מרים את התהליך עם היחס הנמור ביותר עד שהיחס שלו עולה מעל היחס של התחרה הקרוב ביותר. אז התהליך עם היחס הנמור הבא מבחר לרוץ הבא.

אלגוריתם תזמון זה מספק פתרון מעשי לבעה של ההוגנות של אלגוריתם Round Robin. הוא נותן עדיפות לתהליכיים אינטראקטיביים, ובבטיח שככל התהליכיים מקבלים חלקם ההוגן מזמן ה-CPU.

עם זאת, ישנו כמה חסרונות לאלגוריתם זה:

**א. זמן המתנה גבוהה:** זמן המתנה יכול להיות גבוה מאוד והאלגוריתם תמיד בוחר את התהליך עם היחס הנמור ביותר לריצה הבאה.

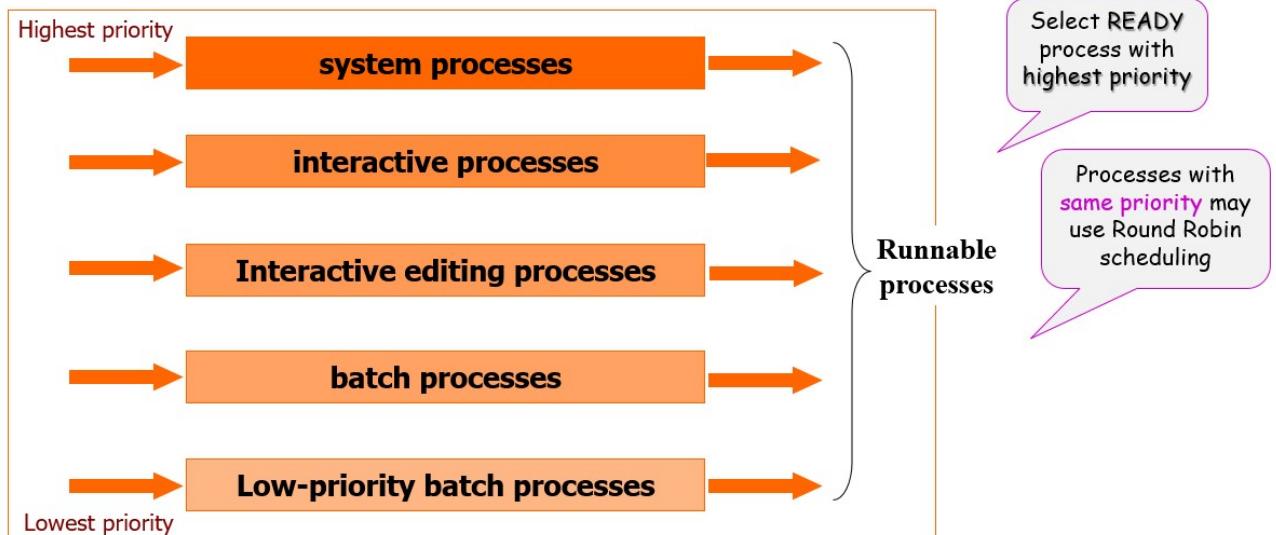
**ב. תהליכיים חדשים בחסרון:** תהליכיים חדשים תמיד יהיו בחסרון לעומת הישנים מאחר שהם השתמשו בפחות זמן CPU.

**ג. אין התחשבות בעדיפויות:** האלגוריתם אינו מתחשב בעדיפויות של התהליכיים.

**ד. אפשרות להרעה:** ישנה אפשרות להרעה. אנחנו לא רוצים שימוש משולש יraig' שהתהליך מורעב.

ניתן להפחית את החסרונות האלה על ידי שימוש באסטרטגיות מסוימות, כמו להקצות את ערך התהליך האב לתהליך חדש או להגדיר ערך ממוצע. עם זאת, פתרונות אלה עשויים לא להיות אופטימליים ועשויים להוביל לביעיות אחרות. לכן, נדרש שיקול דעת והתאמה דקה בעת מימוש המזומנים המובטח (חלוקת הוגנת).

# Multi-Level Scheduling



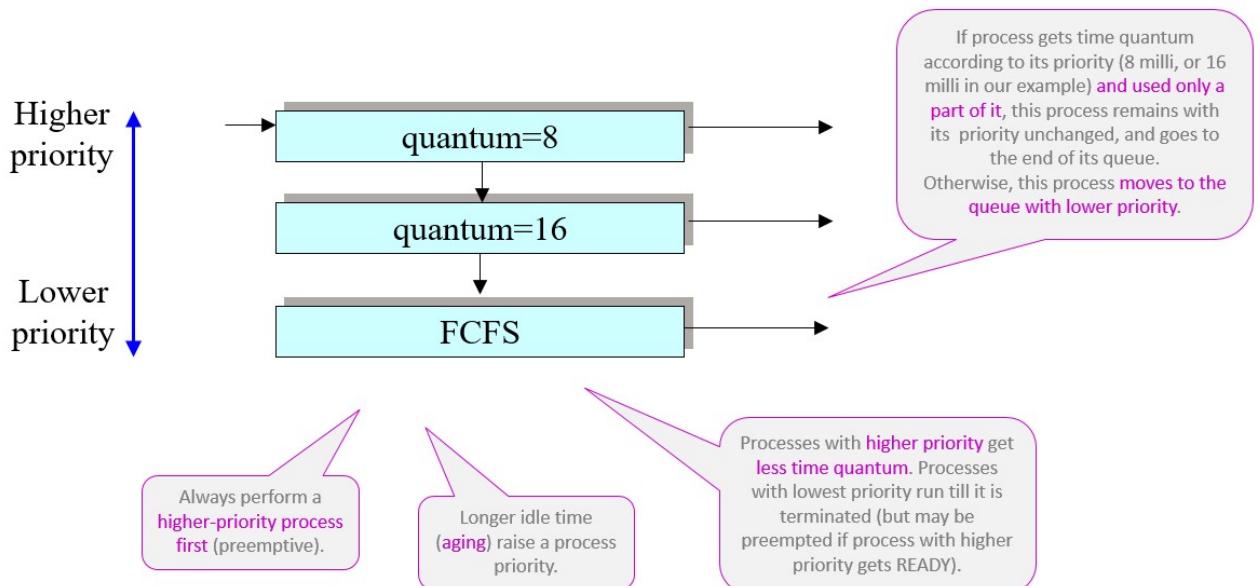
Disadvantage ? Starvation...      How to fix ? Dynamic priority !

## Dynamic multi-level scheduling

- each process has a **base** priority
- increase** priorities of **waiting** processes (at each clock tick)
- increase** priorities of **I/O-bound** processes
- decrease** those of **CPU-bound** processes

prevent high priority processes from running indefinitely

# Multi-level priorities example :Feedback Scheduling



## זמן ברמות מרובות

זמן ברמות מרובות הוא שיטת זמן בה התהליכים ממוקמים לפי העדיפויות שלהם. תהליכי מערכות (כמו NetworkManager לדוגמה) מקבלים את העדיפות הגבוהה ביותר, בעוד שתהליכי משתמשים עם עדיפות נמוכה מקבלים את העדיפות הנמוכה ביותר.

כל הבחירה בשיטה זו הוא פשוט: בחר את התהליך עם העדיפות הגבוהה ביותר. אם יש מספר תהליכים עם אותה עדיפות, משתמשים בשיטת ה-Round Robin לבחירת התהליך להרצה.

## זמן ברמות מרובות דינמי

בזמן ברמות מרובות דינמי, כל תהליך מתחילה עם עדיפות בסיסית. המתזמן ידריל את עדיפות התהליך אם מתקיים אחד מהתנאים הבאים:

- התהליך נמצא במצב המתנה. במקרה זה, המתזמן ידריל את עדיפותו בכל פעם שהשעון מסמן טיק.
- התהליך הוא bound-OI, כלומר מבלה את רוב הזמן בהמתנה לפעולות קלט/פלט להסתויים.

מצד שני, המתזמן מוריד את עדיפות התהליכים שהם CPU-bound, תהליכים שמבלים את רוב הזמן ביצוע הוראות במקום להמתן לפעולות קלט/פלט.

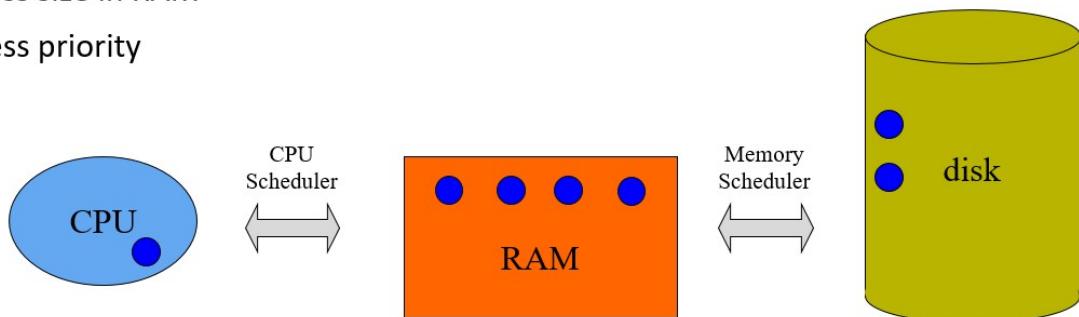
התאמנה דינמית זו של עדיפות מבטיחה שתהליכי bound-OI, שלאrob צריכים להתמכה עם משתמשים או מערכות אחרות, מקבלים גישה מהירה ל-CPU. במקביל, היא מונעת מהתהליכים CPU-bound להשתלט על ה-CPU, ובכך שומרת על איזון במערכת.

חשוב לציין שהמתזמן חייב להימנע מהורדת עדיפות של תהליכי bound-CPU יתר על המידה, שכן גם הם זוקקים לזמן CPU כדי להשלים את משימותיהם. המפתח הוא למצוא איזון ששמסר את המערכת תגובתיות ומבטיחה הקצאת זמן CPU הוגנת לכל התהליכים.

במערכת עם מספר רמות עדיפות, המתזמן בדרך כלל בוחר את התהיליך מהרמה עם העדיפויות האBOVEהו ביותר. אם יש מספר תהיליכים באותו הרמה הזו, המתזמן משתמש בשיטה כמו Round Robin לבחירת התהיליך להרצה. שיטה זו מבטיחה שימושים בעלות עדיפות גבוהה מטופלות קודם, משפרת את תגובתיות המערכת ואת הייעילות שלה.

## 2-Level Scheduling

- ❑ **CPU (low-level) scheduler** – to schedule CPU
  - elapsed time since process swapped-out
  - swapping-out time (dirty process needs more time to be swapped-out)
  - process size in RAM
  - process priority
- ❑ **Memory (high-level) scheduler** – to schedule swap-in / swap-out



### זמן ב-2 רמות

מתזמן low-level: לזמן המעבד.

מתזמן high-level: לזמן החלפות בזיכרון.

המתזמן ב-high level אחראי על שמירת הזמן מאוז שתהיליך הוחלף בזיכרון, זמן להחלפה, גודל התהיליך בזיכרון והעדיפויות של התהיליך.

## Scheduling: outline

- ❑ Scheduling criteria
- ❑ Scheduling algorithms
  - ❑ Unix scheduling
  - ❑ Linux scheduling
  - ❑ Windows scheduling

## Case Study: Unix, Linux, Windows

### Unix Scheduling

#### ❑ 2-level scheduling (Memory and CPU)

#### ❑ Queue for each priority

- user processes have positive priorities
- kernel processes have negative priorities  
**(lower number means higher priority)**

Processes waiting for IO event get high priorities.

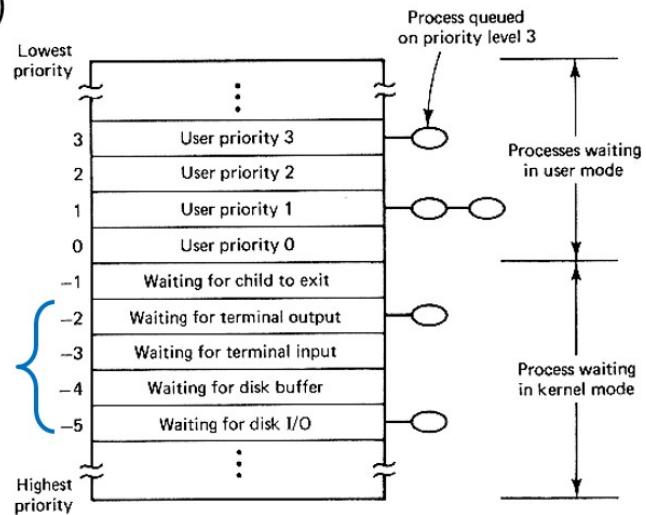


Fig. 7-16. The UNIX scheduler is based on a multilevel queue structure.

### Unix Scheduling

#### ❑ Pick process from highest (non-empty) priority queue

- use RR inside each queue

#### ❑ Run it for one time quantum (usually 100 milli)

#### ❑ Increment CPU usage counter every clock tick

#### ❑ Every second, recalculate priorities:

- CPU usage  $\leftarrow$  CPU usage / 2
- new priority  $\leftarrow$  base priority + CPU usage + nice

fast switch from CPU-bound phase to IO-bound phase, and vice versa

Example: suppose CPU-bound process P that enters to its interactive phase. Suppose P already used 1000 milli CPU time. This value would impact on P's priority for a long time, and we would like to turn P to get priority as other interactive processes do. If we divide each time its CPU time usage by 2, P's CPU time would get smaller exponentially: 500 milli, 250 milli, and so on. If P uses only small CPU number of time units, then its priority would be high very fast.

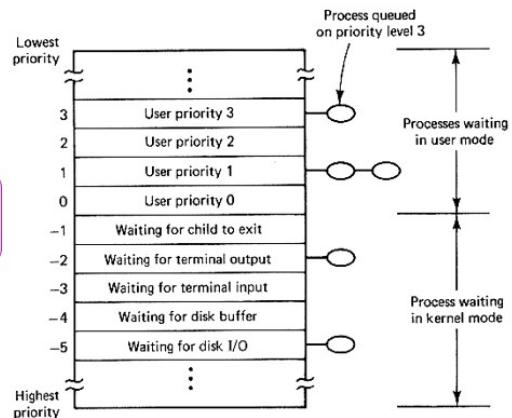


Fig. 7-16. The UNIX scheduler is based on a multilevel queue structure.

# Unix Scheduling

- ❑ Pick process from highest (non-empty) priority queue
  - use RR inside each queue
- ❑ Run it for one time quantum (usually 100 milli)
- ❑ Increment CPU usage counter every clock tick
- ❑ Every second, **recalculate priorities**:
  - CPU usage  $\leftarrow$  CPU usage / 2
  - new priority  $\leftarrow$  base priority + CPU usage + nice

Users with admin permissions may decrease process priority, otherwise user may only decrease 'nice' parameter.

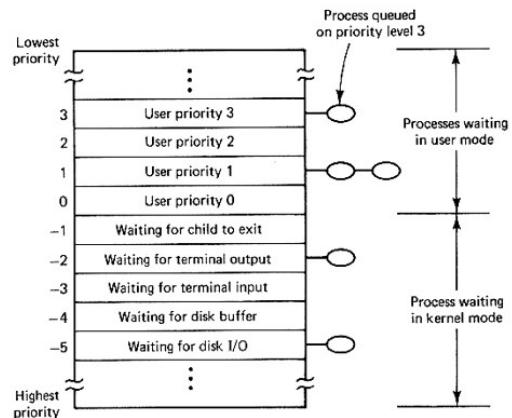


Fig. 7-16. The UNIX scheduler is based on a multilevel queue structure.

## Priority Calculation of Unix

$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{2} + \frac{\text{GCPU}_k(i-1)}{4 \times W_k}$$

$$\text{CPU}_j(i) = \frac{U_j(i-1)}{2} + \frac{\text{CPU}_j(i-1)}{2}$$

$$\text{GCPU}_k(i-1) = \frac{GU_k(i-1)}{2} + \frac{\text{GCPU}_k(i-1)}{2}$$

self-read

UNIX tries to be fair to different users groups - if user A has lots of processes, and user B has only few of them, then A would not get much more CPU time than B.

- $P_j(i)$  = Priority of process  $j$  at beginning of interval  $i$   
 $\text{Base}_j$  = Base priority of process  $j$   
 $U_j(i)$  = Processor utilization of process  $j$  in interval  $i$   
 $GU_k(i)$  = Total processor utilization of all processes in group  $k$  during interval  $i$   
 $\text{CPU}_j(i)$  = Exponentially weighted average processor utilization by process  $j$  through interval  $i$   
 $\text{GCPU}_k(i)$  = Exponentially weighted average total processor utilization of group  $k$  through interval  $i$   
 $W_k$  = Weighting assigned to group  $k$ , with the constraint that  $0 \leq W_k \leq 1$  and  $\sum_k W_k = 1$

## זמן ב-Unix

זמן Unix פועל על מערכת של שני שלבים, שהוא יחסית פשוטה. הוא שומר ערך עדיפות, שיכל להיות חיובי, שלילי, או אפס. לתחביבים של משתמש יש עדיפות חיובית, בעוד שלתחביבים של היררכיה יש ערכאים שליליים. Unix, עדיפות גבוהה יותר מתאימה לערך מספרי קטן יותר, מה שאומר שתחביבי היררכיה מקבלים עדיפות על תחביבי המשתמש.

ישנו 2 סוג תחביבים:

1. **תחביב משותם:** אלה הם תחביבים שממתינים בזמן ריצה וקיים במרחב המשתמש.

**תהליכי קרNEL:** אלה הם תהליכי שמתנים לזמן ריצה וקיימים במרחב הקרNEL. לדוגמה, תהליכי שמתנים 2. יש עדיפות אבואה O/I לאירוע.

כל הבחירה הוא פשוט: בחר תור שאינו ריק עם העדיפות האבואה ביותר. אם יש מספר תהליכי שותפים באותו עדיפות, משתמשים בשיטת Round Robin כדי לבחור תהליך מהතור. התהליך שנבחר אז מורץ למשך קוונטום אחד.

המתזמן של Unix מגדיל את מונה שימוש ה-CPU של תהליך עם כל TICK של השעון ומחשב מחדש את העדיפויות בכל שנייה. הוא עושה זאת על ידי חלקת מدد שימוש ב-CPU ב-2 והוספה התוצאה לעדיפות הבסיסית ולערך 'נחמד' מסוים.

שיטה זו מבטיחה שתהליכי שהם CPU-BOUNDED מוענסים בעדיפות אבואה יותר. במקרים אחרים, תהליכי שוטרים הרבה זמן CPU יהיו להם עדיפות מוגברת, מה שהופך אותם לפחות סבירים להיבחר לתזמון. שיטה זו עוזרת לאזן את העומס בין תהליכי מקושרים ל-CPU ותהליכי מקושרים ל-O/I, ובבטיחה שאף סוג תהליך אחד לא ממונפה על ה-CPU.

## Scheduling: outline

- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux scheduling
- Windows scheduling

# Linux: three classes of threads

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

## Real-time FIFO threads

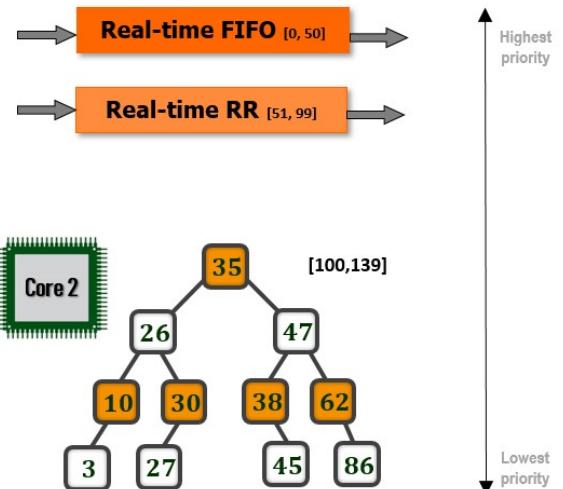
- run till terminated or preempted by Real-time FIFO thread with higher priority

## Real-time Round-Robin threads

- run for time quantum, then preempted by other RR thread

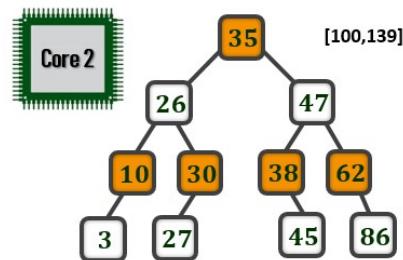
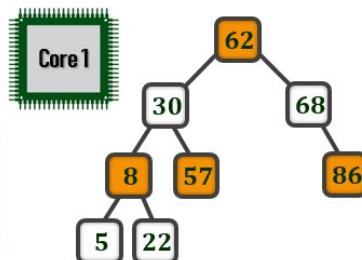
## Other threads

- scheduled by **Completely Fair Scheduler (CFS)**



**Affinity scheduling** -  
thread is strongly  
preferred to be run on  
same core if possible.

Each CPU core manages its own Red-Black (balanced search) tree. Once in some (long) period of time OS executes cores load balancing, by moving threads between cores' trees if needed, so that each core has equal number of threads to run.

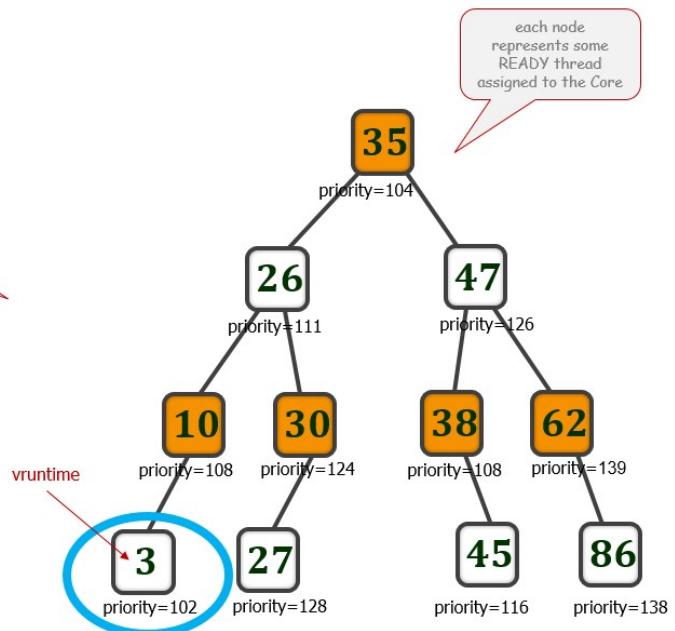


# Linux: Completely Fair Scheduler (CFS)

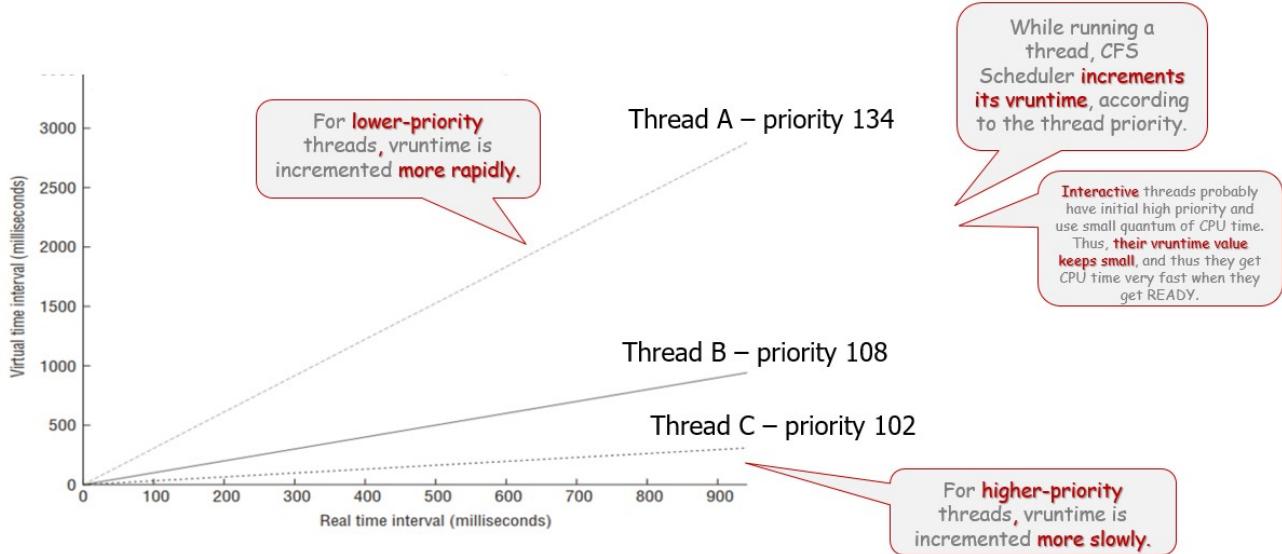
All threads in the tree have priorities 100 – 139. **CFS Scheduler does not change threads priorities** but only change vruntime values. Thread priority may be changed by user application, by using special system call.

Nodes' values are called **vruntime** (virtual run time) – CPU-usage time. This value **increases** during threads life according to their priority.

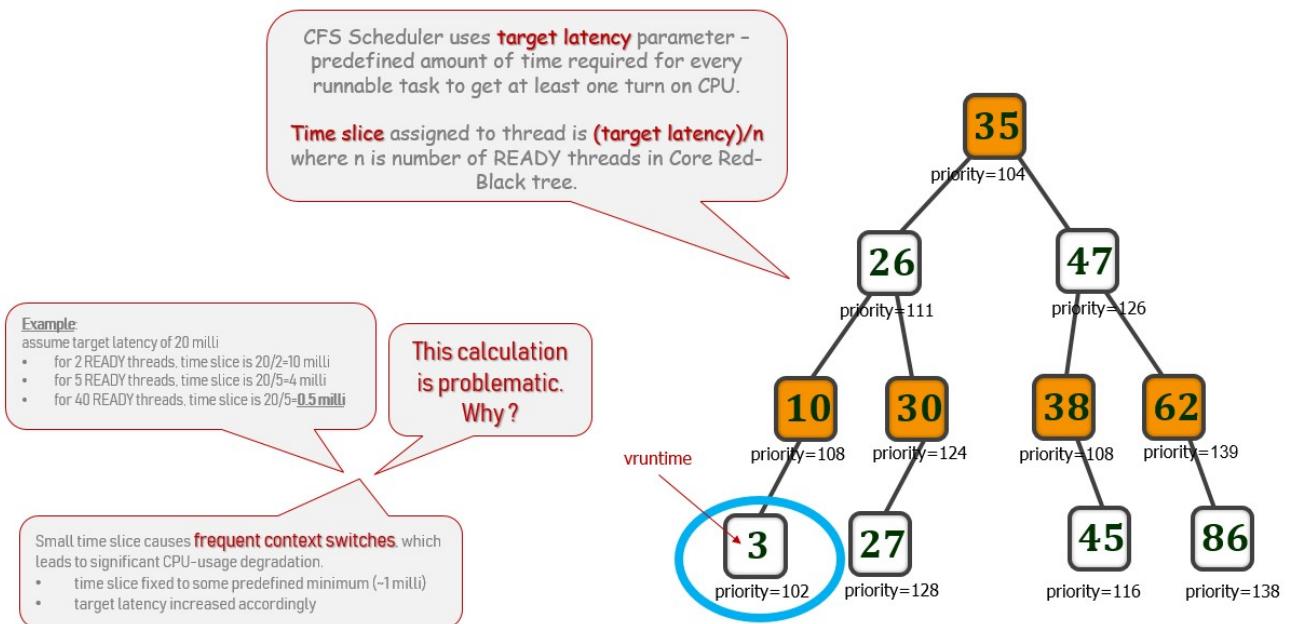
Schedule thread with **minimum vruntime**.



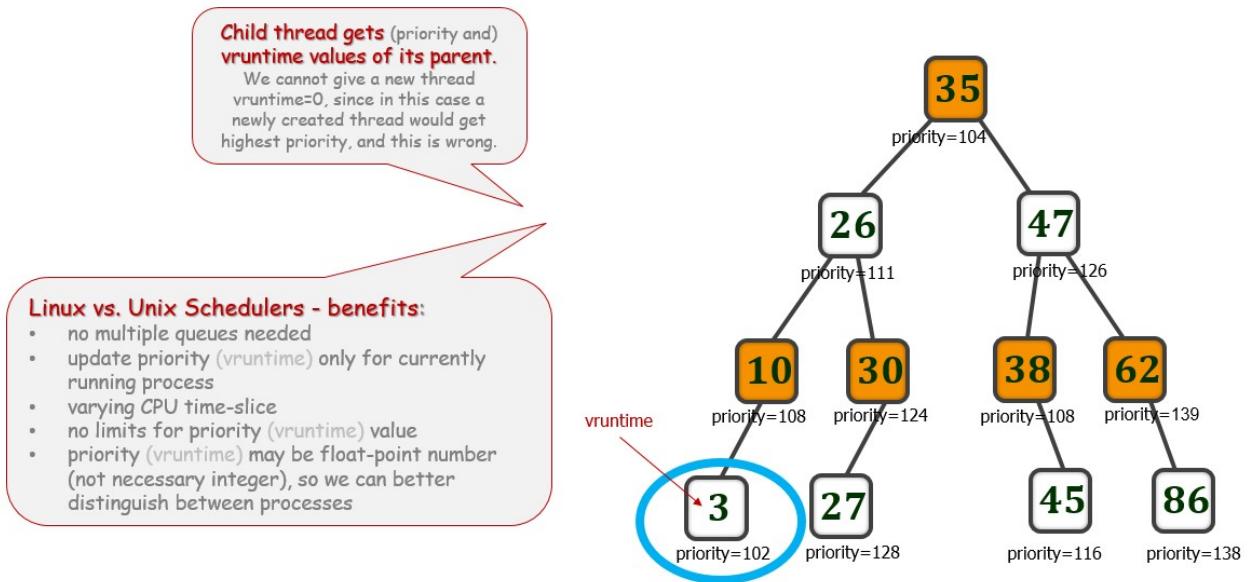
# Linux: Completely Fair Scheduler (CFS)



# Linux: Completely Fair Scheduler (CFS)



# Linux: Completely Fair Scheduler (CFS)



## זמן תהליכי לינוקס

לינוקס, בין היתר, תומכת במערכות זמן אמת. מערכות אלו מעניקות עדיפות בזמן כפרמטר קריטי, במיוחד במערכות ניהול תהליכי תעשייתיים. במקרים כאלה, המערכות חיבוט לאסוף מידע על תהליכי הייצור ולהשתמש במידע זה כדי להפעיל מכונות בפועל, דבר חדשני שמיירה על לוחות זמניים. לעומת זאת, מערכת זמן אמת חיבת המידע לאירועים תוך הזמן הנדרש.

הישויות המרכזיות בפעולה הן תהליכי. כל תהליך מריצ' תוכנית אחת ובהתחלה יש לו ת'ריד אחד שליטה בו. לאחר תחילת הביצוע, הוא יכול ליצור ת'רידים נוספים. ת'ריד (אם נקרא משימה) הוא הישות הבסיסית ביותר שנitin לתזמן. הזמן מודד כמספר טיקים של שעון. תדר השעון משתנה מוגדרת לאחסנה, אך היום הוא מוגדר להיות 500 הרץ, 250 הרץ, או אפילו 1 הרץ. כדי למנוע מחזורי מעבד מבזבזים בפסיקות שעון, ניתן להגדיר את הליבה להיות חסרת טיקים של שעון.

כמו בכל מערכות יוניקס אחרות, לינוקס משיכת ערך nice לכל ת'ריד, שמוגדר בהתחלה להיות 0 וניתן לשנות אותו באמצעות קריית מערכת מיוחדת.

לינוקס תומכת במערכות זמן אמת על ידי סיווג ת'רידים לפי סוגים. לכל סוג יש "מחלקה" משלה עם אלגוריתם זמןון משלה כדי למנוע תחרות של ת'רידים מרמות עדיפות שונות על זמן מעבד.

**ת'רידים של FIFO בזמן אמת:** אלו הם ת'רידים בעלי עדיפות אבואה ולכן הם רצים בסדר FIFO. כל ת'ריד בקבוצה זו יירוץ עד שישים את הריצה שלו, או שייחלף על ידי ת'ריד אחר מאותה הקבוצה אף עם עדיפות אבואה יותר.

**ת'רידים של Round-Robin בזמן אמת:** אלו הם ת'רידים בעלי עדיפות נמוכה יותר ורצים ב-round robin. כל ת'ריד כזה רץ בזמן קוונטום זמן אחד ואז מוחלפים בת'רידים אחרים מאותה קבוצה.

**הת'רידים האחרים מתזמינים על ידי אלגוריתם CFS:** הרעיון העיקרי הוא שימוש בעץ אדום-שחור (שקלול לעץ AVL) כתור שעתקה. המשימות, כלומר הת'רידים, מסודרים בעץ לפי כמות הזמן שהם השתמשו במעבד, שנקרוא

אם ה-vruntimes, כאשר הזמן נמדד באופן כללי ביחידות של ננו-שניות.

באופן כללי, ניתן לתאר את האלגוריתם כך: בחר את התהיליך שקיבל את הזמן הכי פחות לריצה (בדרך כלל הקודקוד השמאלי ביותר בעץ) והרצ אותו. מעת לעת, האלגוריתם מגדיל את ערך ה-vruntimes של התירד, מbasוס על הזמן שכבר רץ, ומשווה אותו לקודקוד השמאלי הנוכחי בעץ. אם עדין יש לו ערך נמוך יותר, הוא ימשיך לרוץ. אחרת, הוא יוכנס לעציו ויתן זמן ריצה לקודקוד השמאלי ביותר בעץ.

כדי להבחין בין העדיפויות של התירדים וערך ה-nice, האלגוריתם משנה את קצב האדילה של ערך ה-vruntimes. בכך, הזמן שבו תהליכיים עם עדיפויות נמוכה רצים חולף מהר יותר מאשר אלה עם עדיפות גבוהה יותר.

בנוסף, לינוקס מבצעת תזמון באפיניטי - משיכת ת'ירדים מסוימים לכל ליבת. יש סיבה לכך - אם אנחנו מניחים ת'ירדים של אותו תהיליך באותה ליבת, אז אנחנו לא צריכים לנוקוט את המטמון. עדין, זה לא הגיוני שליבת 1 תטפל בתהיליך 1. יותר הגיוני לפצל לכמה ליבות.

מה שלינוקס עושה: היא לא מחשבת זמן CPU אמיתי אלא מחשבת זמן מודומה. לכל תהיליך יש עדיפות. ככלומר, אנחנו מוסיפים זמן ריצה לתהיליך כדי להביא לידי ביטוי את ה-priority. ככלומר, לינוקס מעוניינה או מפרגנת לתהיליך לפי ה-priority שלו. ככל שהערך המספרי קטן יותר, הוא מועדף יותר, רק שכן הרכבים לא יכולים להיות שליליים.

ה-target latency - כל כמה מילישניות מתחת לכל התהליכיים הזדמנויות לקבל זמן ריצה כדי שהמשתמש לא ישימ לב שקרה משהו. אם יש חתימות שיכולים לקבל זמן ריצה, אז כל אחד מקבל מספר קבוע (נגיד 20) חלקים. אנחנו משלמים על זה בהחלפת הקשר.

נגיד התירד C רץ באמת 500 מילישניה אבל לפי vruntime קיבל 100 ומהו.

יתרון לינוקס - פחת חישובים. זמני ריצה - שימוש בעצים ולא ברשימות. יוניקס מוגבלת למספר מצומצם של עדיפויות. הבחירה היא מדדייקת יותר - כאן התעדוף משפייע על חישוב הזמן הריצה. ביוניקס התעדוף הוא בעצם הבסיס המינימלי שעלה בסיסו מחושב התעדוף. בלינוקס זה פחת hardcoded.

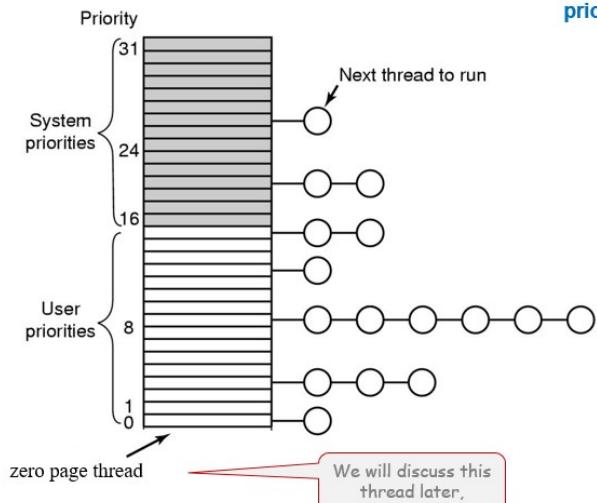
## Scheduling: outline

- ❑ Scheduling criteria
- ❑ Scheduling algorithms
- ❑ Unix scheduling
- ❑ Linux scheduling
- ❑ Windows scheduling

יש עדיפות של תירד ויש עדיפות של תחיל'ר.

## WINDOWS Scheduling

### Priority queues:



Win32  
threads  
priorities

|               | Realtime | High | Above Normal | Normal | Below Normal | Idle |
|---------------|----------|------|--------------|--------|--------------|------|
| Time critical | 31       | 15   | 15           | 15     | 15           | 15   |
| Highest       | 26       | 15   | 12           | 10     | 8            | 6    |
| Above normal  | 25       | 14   | 11           | 9      | 7            | 5    |
| Normal        | 24       | 13   | 10           | 8      | 6            | 4    |
| Below normal  | 23       | 12   | 9            | 7      | 5            | 3    |
| Lowest        | 22       | 11   | 8            | 6      | 4            | 2    |
| Idle          | 16       | 1    | 1            | 1      | 1            | 1    |

Note that **higher** numeric value means **higher** priority.

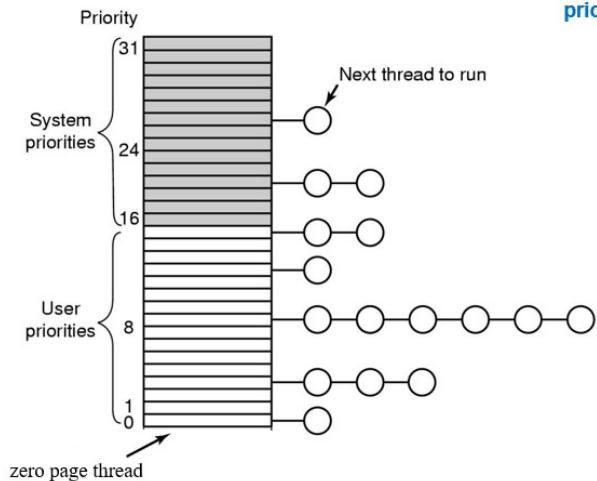
[1,15] - user/system threads  
[16,31] - system threads only

Schedule thread with **maximum** priority.

תירד לא יכול לעבור את התעדוף של תחיל'ר.  
כאן זה הפך מינוקס/לינוקס.

## WINDOWS Scheduling

### Priority queues:



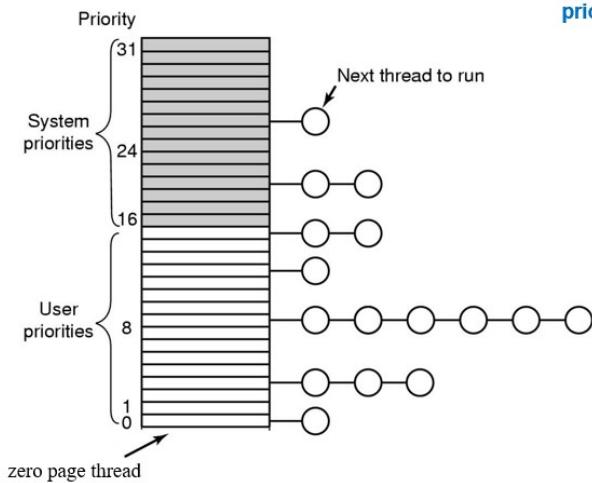
Win32  
threads  
priorities

|               | Realtime | High | Above Normal | Normal | Below Normal | Idle |
|---------------|----------|------|--------------|--------|--------------|------|
| Time critical | 31       | 15   | 15           | 15     | 15           | 15   |
| Highest       | 26       | 15   | 12           | 10     | 8            | 6    |
| Above normal  | 25       | 14   | 11           | 9      | 7            | 5    |
| Normal        | 24       | 13   | 10           | 8      | 6            | 4    |
| Below normal  | 23       | 12   | 9            | 7      | 5            | 3    |
| Lowest        | 22       | 11   | 8            | 6      | 4            | 2    |
| Idle          | 16       | 1    | 1            | 1      | 1            | 1    |

Example of table usage: for process with priority Normal, and for thread with priority Above normal, this thread would be in **priority = 14** queue.

# WINDOWS Scheduling

## Priority queues:



Win32  
threads  
priorities

|               | Realtime | High | Above Normal | Normal | Below Normal | Idle |
|---------------|----------|------|--------------|--------|--------------|------|
| Time critical | 31       | 15   | 15           | 15     | 15           | 15   |
| Highest       | 26       | 15   | 12           | 10     | 8            | 6    |
| Above normal  | 25       | 14   | 11           | 9      | 7            | 5    |
| Normal        | 24       | 13   | 10           | 8      | 6            | 4    |
| Below normal  | 23       | 12   | 9            | 7      | 5            | 3    |
| Lowest        | 22       | 11   | 8            | 6      | 4            | 2    |
| Idle          | 16       | 1    | 1            | 1      | 1            | 1    |

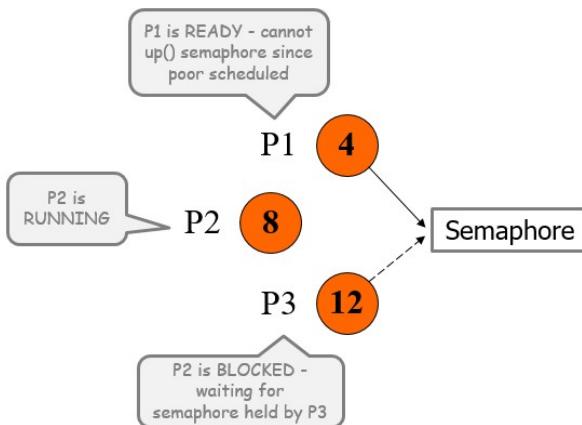
### Priority is dynamically increased:

- Upon IO completion
- Upon received event
- If a thread waits too much - it is moved to priority 15 for two time-quantum

### Priority is dynamically decreased:

- By 1, if time quantum is fully used
- Never below process type base value (according to the table)

## Priority inversion problem



We are given 3 processes -  $P_1$  with priority 12,  $P_2$  with priority 8, and  $P_3$  with priority 4.

$P_1$  is BLOCKED on semaphore,  $P_2$  is RUNNING, and  $P_3$  is READY but do not get CPU time due to its low priority. After  $P_3$  would wait enough time, its priority would be moved to 15 for two time-quantum, and this would let it up a semaphore and release  $P_1$ .



© 2022 מתקנים הפעלה

# Operating Systems

## Lecture 3 – Scheduling

Dr. Marina Kogan-Sadetsky