תרגול 5

מבוא לזימון תהליכים
זימון תהליכי זמן אמת בלינוקס
זימון תהליכים רגילים בלינוקס

TL;DR

- זַמָּן התהליכים (scheduler) הוא הרכיב במערכת ההפעלה שאחראי על בחירת התהליך הבא שירוץ על המעבד.
- אנחנו נלמד, בתור דוגמה, את אלגוריתם הזימון של לינוקס.

זימון תהליכים בלינוקס

SCHED_FIFO, SCHED_RR
אלגוריתם זימון של תהליכי זמן אמת

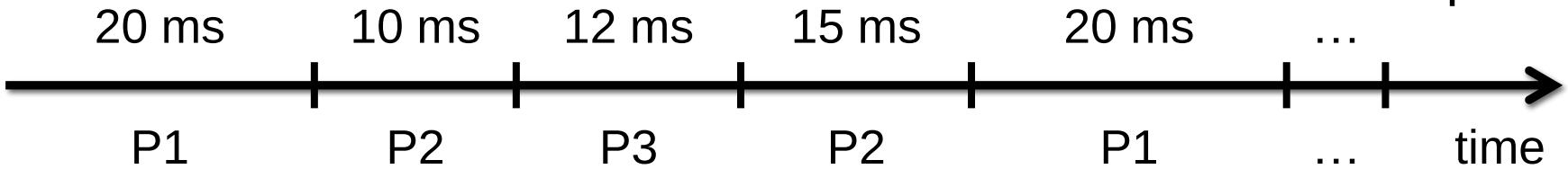
CFS = completely fair scheduler
אלגוריתם זימון של תהליכים רגילים

- אבל לפני שנלמד דוגמה "אמיתית" ומורכבת, נרצה להבין את הגישות הבסיסיות בזימון תהליכים כדי לפתח אינטואיציה.

מבוא לזימון תהליכים

הגדרת הבעיה

- במערכות מחשבים אמיתיות, השמיכה קצרה מדי: בכל רגע מחכים לרוץ יותר תהליכים ממספר המעבדים שיכולים להריץ אותם.
- אין ברירה: מערכת ההפעלה צריכה לחלק את זמן המעבדים בין התהליכים.



- איך כדאי לבחור, בכל נקודת זמן, את התהליך הבא שירוץ על המעבד? וכמה זמן לתת לתהליך הזה?
- אין תשובה "נכונה". צריך למצוא פשרה בין מספר דרישות הנדסיות: הוגנות, אינטראקטיביות ויעילות.

יעילות, אינטראקטיביות, והוגנות

- **יעילות** – תהליכים מסתיימים מהר ככל הניתן.
  - בדרך כלל יעילות מוגדרת באמצעות מדד זמן ההמתנה הממוצע.
- **אינטראקטיביות** – תהליכים מגיבים מהר לפעולות I/O.
  - לא היינו רוצים ששיחת טלפון תהיה מקוטעת כי תהליך חיפוש קבצים רץ ברקע.
- **הוגנות** – אין הגדרה חד-משמעית, אלא יש כמה גישות.
  - למשל: תהליך שהגיע ראשון, ירוץ ראשון על המעבד.
  - למשל: כל התהליכים מקבלים את אותו זמן מעבד.
- שלושת הדרישות סותרות אחת את השנייה, ולכן יש למצוא איזון ביניהן.

5 הנחות על העומס

- נרצה לפתח אלגוריתם זימון בסיסי.
- לשם כך נניח 5 הנחות מפשטות ולא ריאליסטיות על העומס (workload) במערכת:

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מגיעים באותו זמן (t=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

- בהמשך נסיר לאט לאט את ההנחות האלו כדי לפתח אלגוריתם "אמיתי".

מדדי ביצועים (מטריקות)

- כדי להשוות בין אלגוריתמי זימון שונים באופן כמותי, נגדיר מספר מדדי ביצועים.
- המדד הראשון יהיה: "זמן התגובה הממוצע".
- לכל תהליך נגדיר את "זמן התגובה":

  responseTime = terminationTime – arrivalTime
- מכיוון שיש הרבה תהליכים, נמצע את זמן התגובה על פני כולם.

- כרגע, תחת ההנחות שהגדרנו, כל התהליכים מגיעים בזמן 0.
- ← זמן התגובה == זמן הסיום.

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מגיעים באותו זמן (t=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתם FCFS

- FCFS = first come, first served
- נקרא גם: FIFO = first in, first out
- אופן פעולה: תור הוגן.

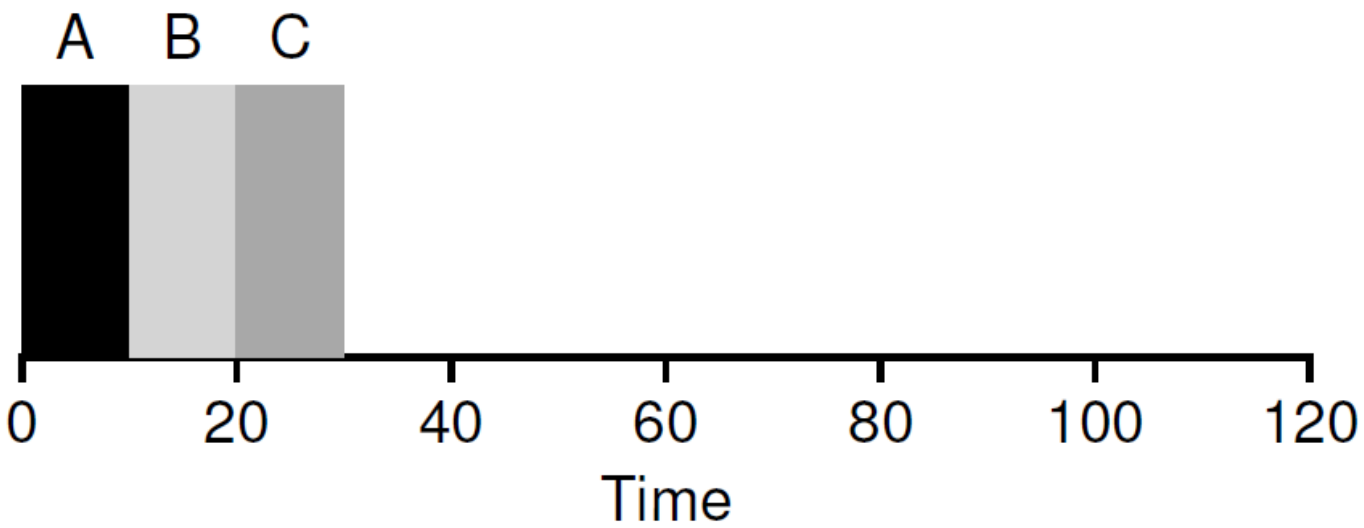
- לדוגמה:
- נניח כי 3 תהליכים A,B,C הגיעו בזמן t=0 למערכת.
  - נניח ש–A הגיע טיפה לפני B, שהגיע טיפה לפני C.
- בנוסף נניח כי זמן הריצה של כל אחד מהתהליכים הוא 10 שניות.

- איך נראה הזימון? מה זמן התגובה הממוצע?

דוגמת FCFS

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מגיעים באותו זמן (t=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

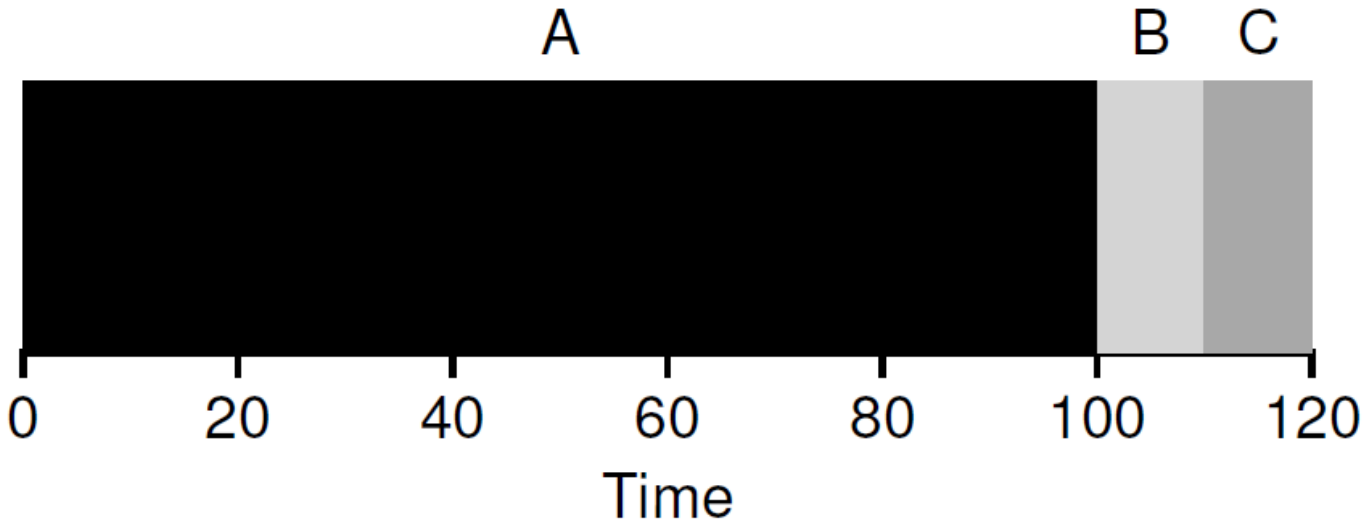
averageResponseTime = (10 + 20 + 30) / 3 = 20



- כעת נסיר את הנחה 1 ("כל התהליכים רצים למשך אותו זמן").
  ← לכל תהליך זמן ריצה משלו.
- תוכלו לחשוב על דוגמה שבה FCFS אינו יעיל?

אפקט השיירה (convoy effect)

averageResponseTime = (100 + 110 + 120) / 3 = 110



- אלגוריתם FCFS עלול לסבול מ"אפקט השיירה": מצב שבו תהליך אחד ארוך מעכב הרבה תהליכים קצרים.

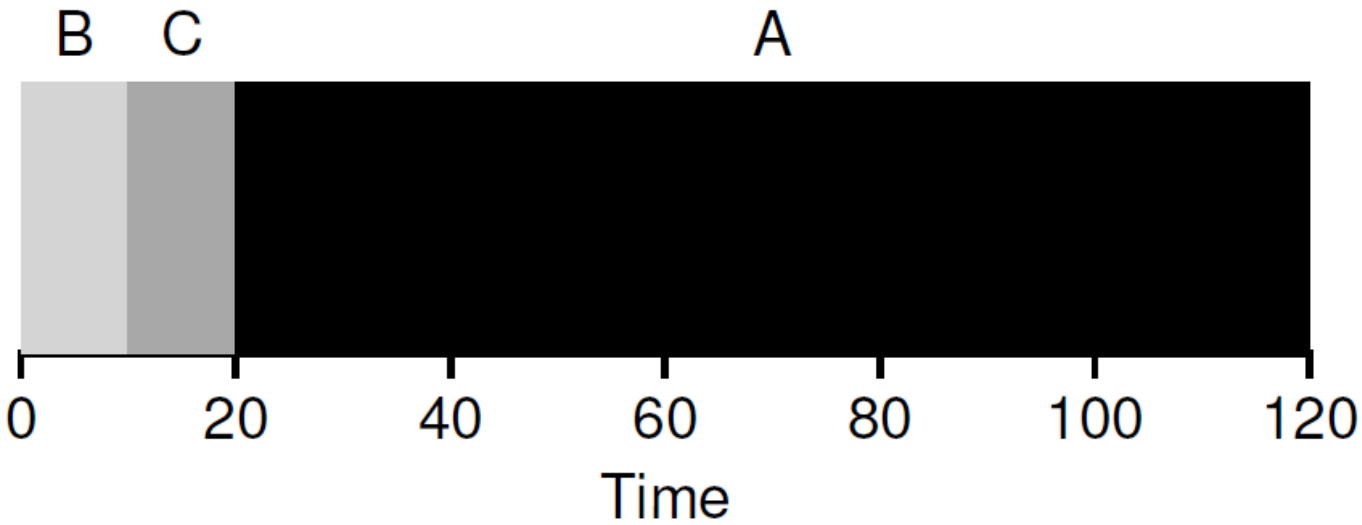
1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. כל התהליכים מגיעים באותו זמן (t=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתם SJF

- SJF = shortest job first
- אופן פעולה: השם אומר הכול.

- בדוגמה האחרונה נקבל:

averageResponseTime = (10 + 20 + 120) / 3 = 50



1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. כל התהליכים מגיעים באותו זמן (t=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

אופטימליות של SJF

- תחת ההנחות שהגדרנו בהתחלה (פחות הנחה 1 עליה כבר וויתרנו) ניתן להוכיח כי SJF הוא האלגוריתם האופטימלי במדד זמן תגובה ממוצע.

- האינטואיציה מאחורי SJF היא לתת עדיפות ללקוחות (== תהליכים) קטנים כדי שיהיו מרוצים (== זמן תגובה נמוך).
  - אנלוגיה לקופת "עד 10 פריטים" בסופר.

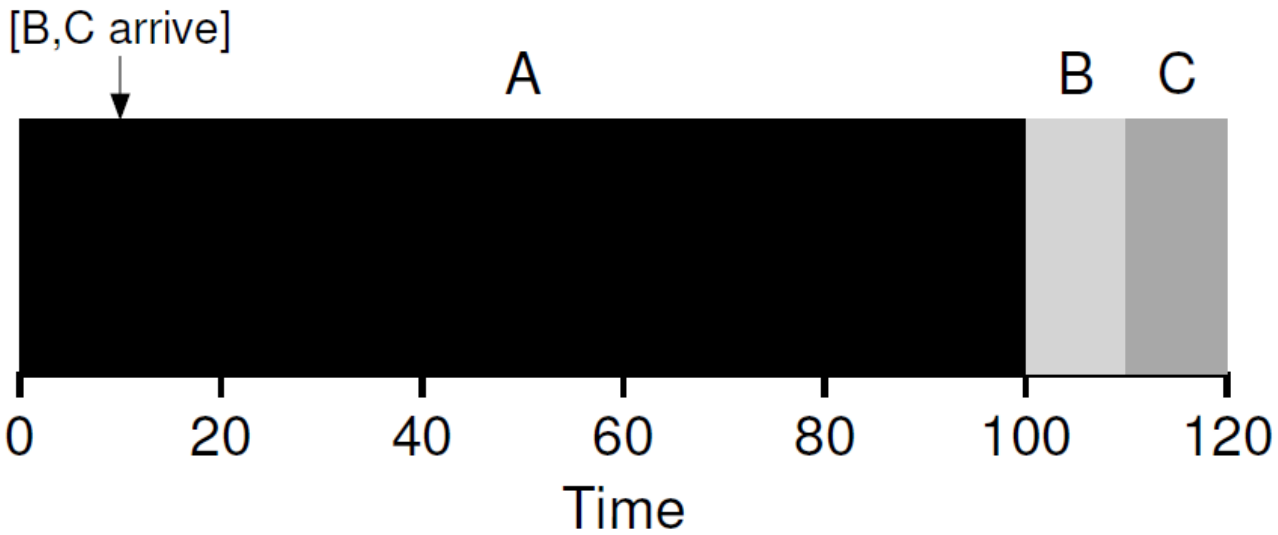
- כעת נסיר את הנחה 2 ("כל התהליכים מגיעים באותו זמן t=0").
  ← תהליכים יכולים להגיע בכל זמן t>0.
- תוכלו לחשוב על דוגמה שבה SJF אינו יעיל?

1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

שוב אפקט השיירה...

- תהליך A מגיע בזמן t=0 ורץ למשך 100 שניות.
- תהליכים B,C מגיעים בזמן t=10 ומבקשים לרוץ 10 שניות כל אחד.
- שוב תהליכים קצרים מתעכבים מאחורי תהליך ארוך.

avgResponseTime = (100 + 100 + 110) / 3 = 103.33



1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. ~~אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.~~
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

הפתרון: הפקעה

- כדי לפתור את הבעיה, נסיר את הנחה 3 ("אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות").
  ← מערכת ההפעלה יכולה להפקיע את המעבד מתהליך רץ, כלומר: לעצור את התהליך הנוכחי ולקרוא לתהליך אחר במקומו.

- המעבר בין התהליכים נקרא "החלפת הקשר".

- תזכורת: המנגנון שמאפשר הפקעה הוא פסיקות שעון.

1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. ~~אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.~~
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתם SRTF

- SRTF = shortest remaining time first
- נקרא גם: STCF = shortest time to completion first

- אופן פעולה: כמו SJF, אבל עם הפקעות.
- בכל פעם שתהליך חדש מגיע למערכת, SRTF מחשב למי מבין התהליכים (כולל התהליך החדש) נותר הכי פחות זמן לרוץ, ובוחר את התהליך הזה לריצה.

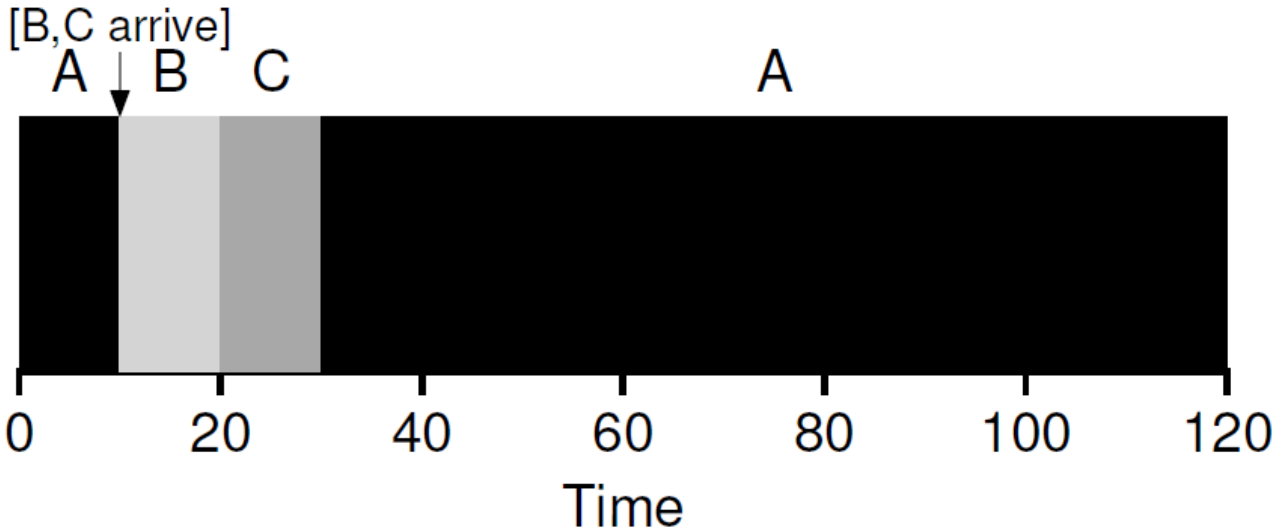
- תחת ההנחות החדשות, ניתן להוכיח כי SRTF אופטימלי במדד זמן התגובה הממוצע.
  - בתוספת הנחה כי זמן החלפת הקשר הוא אפסי.

דוגמת SRTF

1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. ~~אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.~~
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

- תהליך A מגיע בזמן t=0 ורץ למשך 100 שניות.
- תהליכים B,C מגיעים בזמן t=10 ומבקשים לרוץ 10 שניות כל אחד.

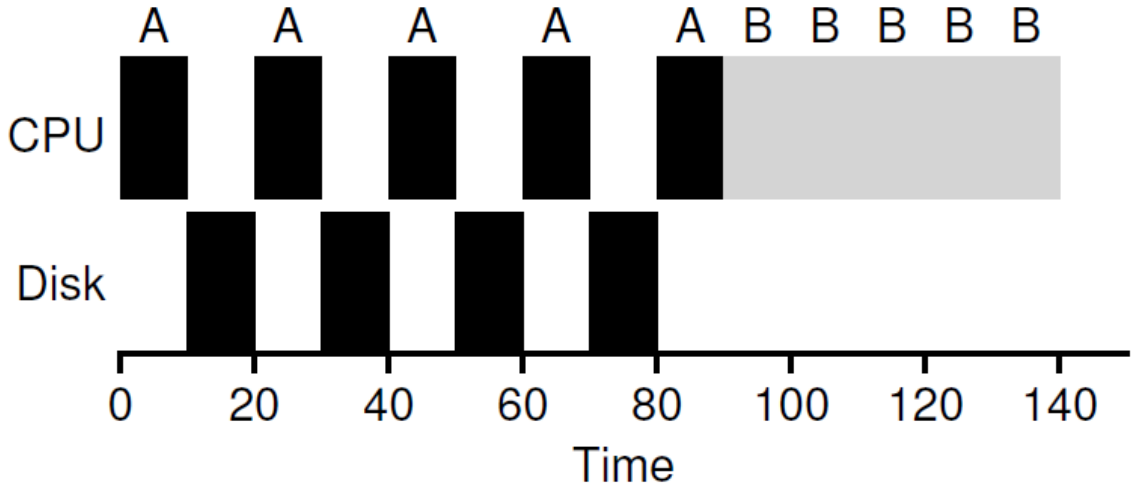
avgResponseTime = (10 + 20 + 120) / 3 = 50



1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. ~~אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.~~
4. ~~התהליכים משתמשים רק במעבד ולא מבצעים I/O.~~
5. זמן הריצה של כל התהליכים ידוע מראש.

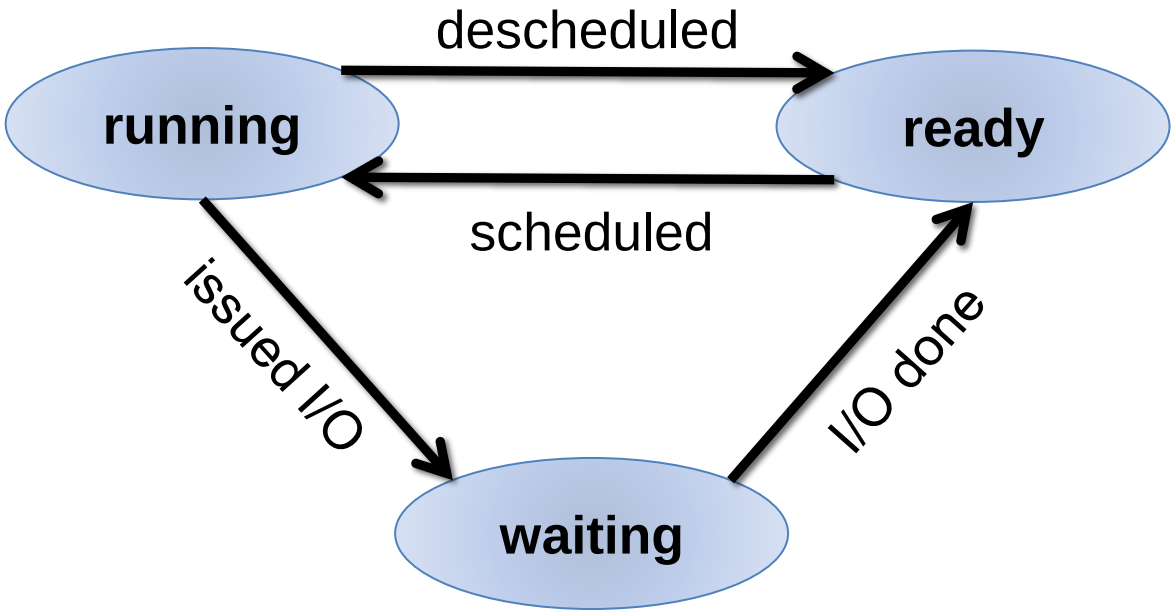
שילוב התקני I/O

- כעת נסיר גם את הנחה 4 (" התהליכים משתמשים רק במעבד ולא מבצעים I/O").
  ← התהליכים ניגשים להתקני I/O, לדוגמה דיסק או כרטיס רשת.
  - גישה להתקני I/O אורכת זמן רב (במונחי מעבד) – מספר מילישניות או יותר.
- תהליך שממתין ל–I/O לא משתמש במעבד – בעיית יעילות.



ניצול טוב יותר של המשאבים

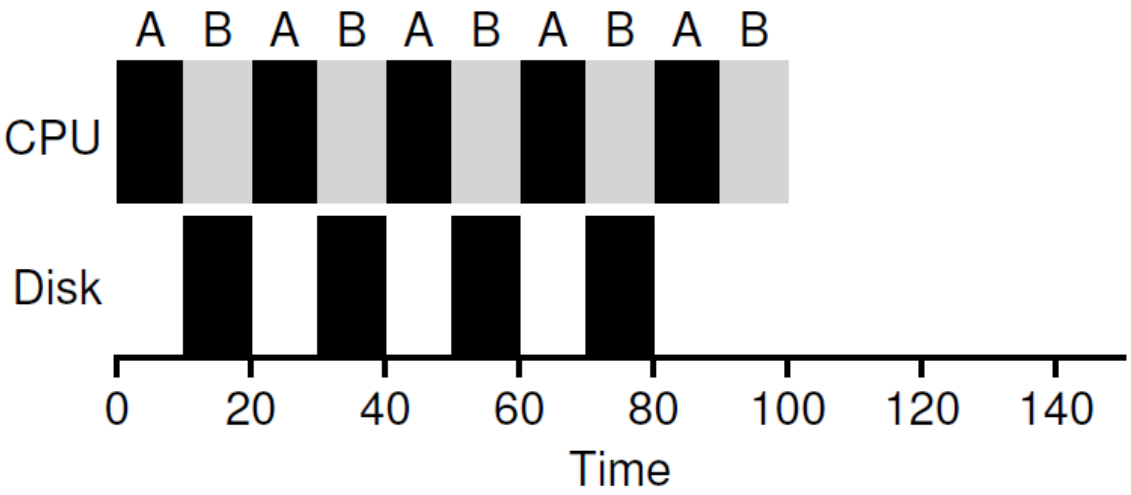
- כאשר התהליך הנוכחי מבקש I/O – אלגוריתם הזימון יעביר אותו למצב המתנה ויזמן תהליך אחר במקומו.
- כאשר ההתקן מסיים את פעולתו ושולח פסיקה למעבד – אלגוריתם הזימון יחזיר את התהליך למצב מוכן לריצה (לא בהכרח יריץ אותו).



1. ~~כל התהליכים רצים למשך אותו זמן.~~
2. ~~כל התהליכים מגיעים באותו זמן (t=0).~~
3. ~~אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.~~
4. ~~התהליכים משתמשים רק במעבד ולא מבצעים I/O.~~
5. זמן הריצה של כל התהליכים ידוע מראש.

דוגמה נוספת של SRTF

- תהליך A רץ 50ms בסך הכל.
  בכל 10ms הוא מבצע I/O שאורך גם כן 10ms.
- תהליך B גם רץ 50ms בסך הכל, אבל לא מבצע I/O בכלל.
- הגישה המקובלת היא להתייחס לכל תת-משימה של A כאל תהליך עצמאי באורך 10ms. איך ייראה זימון תחת אלגוריתם SRTF?



נשארנו רק עם הנחה 5...

- "זמן הריצה של כל התהליכים ידוע מראש."

- ההנחה הזו הגיונית במקרים מסוימים.
- למשל, במקרה של batch scheduling כפי שלמדתם בהרצאה:
  - הרבה משתמשים עובדים על מחשב-על (supercomputer) בעל הרבה ליבות.
  - המשתמשים שולחים "עבודות" (jobs) לתור הריצה.
  - אלגוריתם הזימון משבץ את העבודות על הליבות הפנויות.
  - המשתמשים נדרשים לספק הערכה לזמן הריצה של העבודות שהם שולחים.
  - עבודה שחורגת מזמן הריצה שהו[illegible] – נעצרת ע"י אלגוריתם

האם כדאי למשתמש לספק הערכה גבוהה כדי להבטיח שהעבודה שלו תסתיים בצורה תקינה?

Batch scheduling

- אלגוריתמי batch scheduling מניחים את הנחות 3,4,5 שראינו:
  3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסקות.
  4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
  5. זמן הריצה של כל התהליכים ידוע מראש.

- שימו לב: אנחנו הנחנו עבודות סדרתיות, אבל בהרצאה מטפלים גם בעבודות מקביליות (== רצות על מספר ליבות במקביל).
- רוב התוצאות התיאורטיות כבר לא תקפות לעבודות מקביליות.
  - למשל, SJF כבר לא אופטימלי במדד זמן המתנה ממוצע.
- אבל האינטואיציה שקיבלנו בעבודות סדרתיות בדרך כלל נשמרת.

מדד חדש: זמן המתנה

- עד כה למדנו מספר אלגוריתמי זימון והשווינו ביניהם לפי מדד "זמן התגובה":

responseTime = **terminationTime** – arrivalTime

- כעת נציג מדד חדש – "זמן ההמתנה":

waitTime = **startTime** – arrivalTime

  - כמו קודם, המדד יהיה זמן ההמתנה הממוצע על פני כל התהליכים.

- איזה מדד קובע את הביצועים?
- התשובה תלויה בעומס...

נהוג לסווג תהליכים לשני סוגים

תהליך חישובי
CPU Bound

- מעוניין בזמן **תגובה** נמוך.
  - throughput sensitive.
  - דוגמה: סקריפט python שמנתח נתונים ע"י חישובים אלגבריים.
- בדרך-כלל לא מוותר על המעבד מרצונו אלא מופקע.

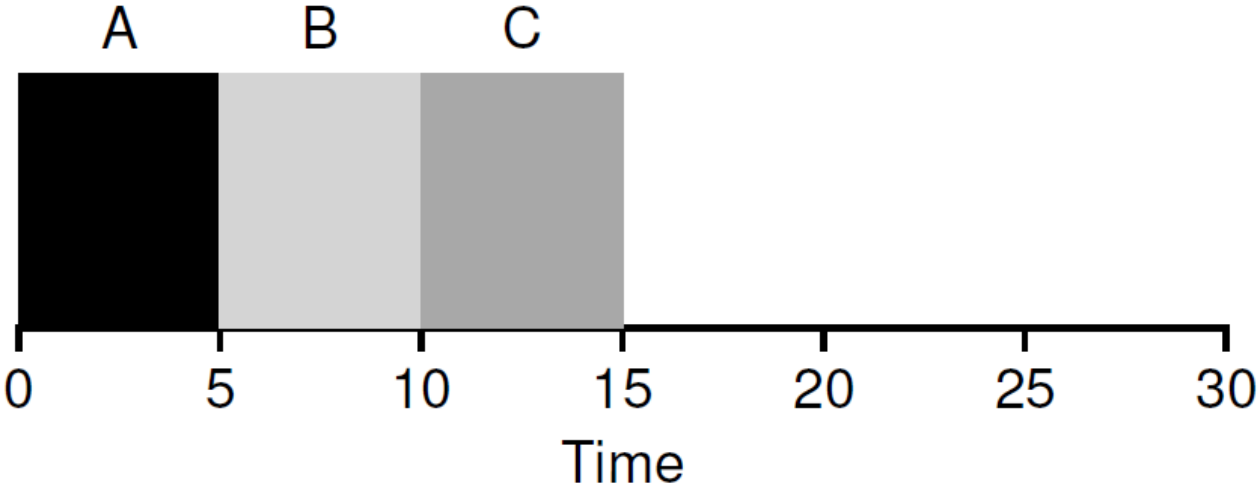
תהליך אינטראקטיבי
I/O Bound

- מעוניין בזמן **המתנה** נמוך.
  - latency sensitive.
  - דוגמה: נגן סרטים שמחליף 60 פריימים בשנייה.
- בדרך-כלל מוותר על המעבד מרצונו אחרי פרק זמן קצר בגלל המתנה לפעולות I/O.

SJF / SRTF לא מצטיינים בזמן ההמתנה

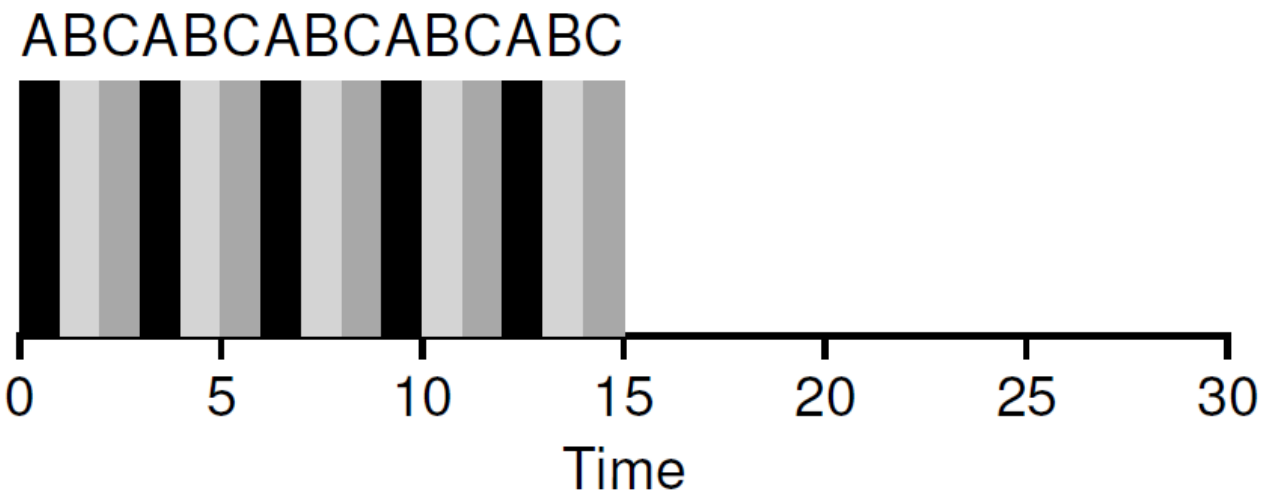
- נניח שלושה תהליכים A,B,C מגיעים באותו הזמן.
- כל תהליך רץ למשך 5 שניות.
- מה יהיה זמן ההמתנה תחת אלגוריתם SJF או SRTF?

$$\text{avgWaitTime} = (0 + 5 + 10) / 3 = 5$$



אלגוריתם Round Robin

- במקום להריץ תהליכים עד לסיומם, האלגוריתם מריץ כל תהליך במשך פיסת זמן מסוימת (time slice או quantum), ואז עובר להריץ תהליך אחר למשך אותה פיסת זמן, וכן הלאה...
- אם נניח כי הקוונטום הוא 1s , אז: avgWaitTime ≈ 2s .
  - לצורך חישוב זמן ההמתנה, נתייחס לכל פיסת זמן כאל תהליך עצמאי באורך 1s אשר מגיע ברגע שפיסת הזמן הקודם הסתיימה.



שיקולים בבחירת הקוונטום

- הקוונטום חייב להיות כפולה של הזמן בין פסיקות שעון.
  - למשל, אם פסיקות שעון מגיעות כל 10ms,
    אז הקוונטום יכול להיות 10ms, 20ms, 30ms ...
- למה עדיף קוונטום נמוך?
  - זמן המתנה נמוך ← יותר אינטראקטיביות.
- למה עדיף קוונטום גבוה?
  - פחות החלפות הקשר ← ביצועים טובים יותר.
- למשל נניח כי הקוונטום הוא 10ms וזמן החלפת הקשר הוא 1ms.
- מה התקורה של אלגוריתם RR במקרה הזה?

פשרה בין זמן תגובה לזמן המתנה

SJF, SRTF

- מנסים להביא למינימום את זמן התגובה הממוצע.
- אבל משיגים זמן המתנה גרוע.

- האלגוריתמים האלו מניחים כי זמן הריצה של כל התהליכים ידוע מראש.

RR

- מנסה להביא למינימום את זמן ההמתנה הממוצע.
- אבל משיג זמן תגובה גרוע.

- האלגוריתם אינו צריך לדעת מראש את זמן הריצה של כל התהליכים.

אלגוריתמי זימון של מערכות אמיתיות (למשל CFS של לינוקס) ינסו לשלב בין שני הסוגים לעיל.

הפסקה

זימון תהליכים בלינוקס

שני סוגי תהליכים בלינוקס

תהליכי זמן-אמת (real-time)

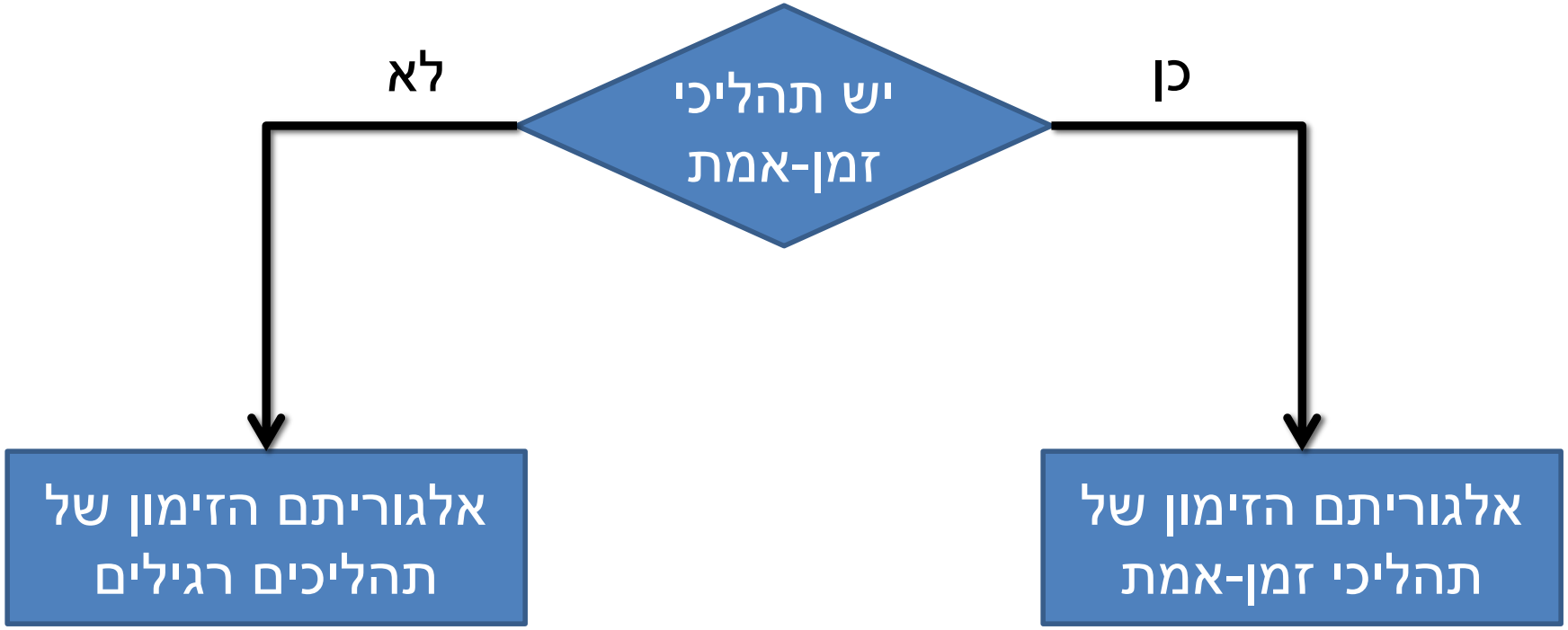
- נדרשים לעמוד באילוצים קשיחים על זמן התגובה, ללא תלות בעומס על המערכת.
  - דוגמה: תוכנת הטייס האוטומטי במטוסים.
- רק משתמש-על (root) יכול להגדיר תהליך כזמן-אמת ע"י שינוי העדיפות שלו.

תהליכים "רגילים" (conventional)

- אינם רגישים לזמן התגובה ויכולים לספוג עיכובים במידה והמערכת עמוסה (כלומר אם רצים עוד הרבה תהליכים במקביל).
  - דוגמה: תהליך שמנתח את המידע שנאסף בטיסה לאחר הטיסה.
  - דוגמה: תוכנית ה-Powerpoint בה אתה רואים שקופית זו.

שני סוגי תהליכים בלינוקס

- לכל אחד מסוגי התהליכים אלגוריתם זימון שונה.
- תהליכים רגילים אינם זוכים לרוץ אם יש תהליכי זמן-אמת מוכנים לריצה (כלומר הנמצאים במצב TASK_RUNNING).



תזכורת: תורי ריצה ותורי המתנה

- התהליכים המוכנים לריצה (מצב TASK_RUNNING) נשמרים במבנה נתונים הקרוי **runqueue** (תור ריצה).

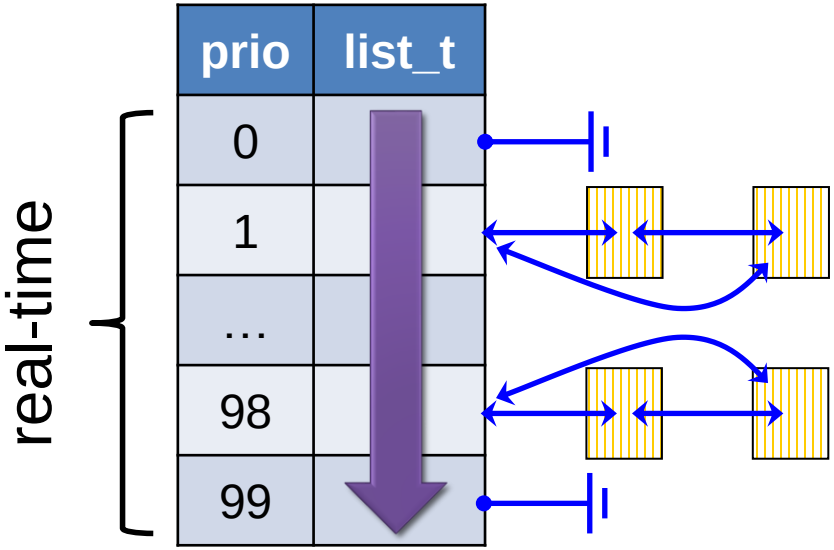
- לכל ליבת מעבד יש תור ריצה (מבנה runqueue) משלה.
- בכל רגע נתון, תהליך יכול להימצא בתור ריצה אחד לכל היותר.
- אלגוריתם הזימון בוחר את התהליך הבא לריצה על המעבד מתוך תור הריצה של אותו מעבד.
  - לינוקס מעבירה תהליכים בין תורי ריצה כדי לאזן עומסים אם היא מזהה שיש הרבה תהליכים במעבד אחד לעומת מעבד שני. נושא זה מעבר לחומר הקורס.

- שימו לב: תהליכים במצבי המתנה (== נמצאים בתורי המתנה) אינם נמצאים בתורי הריצה ואינם קיימים מבחינת אלגוריתם הזימון.

זימון תהליכי זמן אמת בלינוקס

אלגוריתם הזימון של תהליכי זמן–אמת

- כל התהליכים המוכנים לריצה (מצב TASK_RUNNING) נשמרים בתור הריצה: מערך תורים באורך 100, תור לכל עדיפות מספרית.
  - כל תהליך נמצא בתור אחד בלבד.
- התהליכים מזומנים לפי **עדיפות** – תהליכים בעדיפות נמוכה יזומנו רק אחרי שהסתיימו כל התהליכים בעדיפות הגבוהה.
  - מספר גבוה == עדיפות נמוכה.



- האלגוריתם תמיד בוחר לריצה את התהליך שנמצא בראש התור עם העדיפות הגבוהה ביותר.

מדיניות זימון של תהליך

- לכל תהליך זמן-אמת יש מדיניות זימון (scheduling policy): SCHED_FIFO או SCHED_RR .
- נקבעת ע"י המשתמש באמצעות קריאות מערכת sched_setscheduler() .
- **מדיניות הזימון** – תשפיע על כמה זמן ריצה כל תהליך יקבל ואופן עבודת התור.

מדיניויות זימון של תהליכי זמן–אמת

SCHED_FIFO

First in First Out

- זימון לפי סדר הגעה.
- תהליך FIFO **מוותר** על המעבד רק אם:
  - הוא יוצא להמתנה (למשל בגלל I/O) – בעתיד יחזור לסוף התור.
  - הוא קורא לקריאת המערכת sched_yield – עובר מיד לסוף התור (נשאר בתור הריצה).
- תהליך FIFO **מופקע** רק ע"י תהליך זמן–אמת אחר עדיף יותר.

SCHED_RR

Round Robin

- חלוקת זמן בין כל התהליכים המוכנים לריצה בעלי העדיפות הטובה ביותר.
  - כל תהליך מקבל קוונטום (פיסת זמן) בתורו, לפי סדר מעגלי.
- **<u>שימו לב:</u>** תהליכים במדיניות זימון SCHED_RR עלולים "להיתקע" בתור אחרי תהליכים במדיניות זימון SCHED_FIFO.

דוגמה

תהליך	A	B	C	D
זמן הגעה	1	2	2	3
עדיפות	30	31	30	30

- אם כל התהליכים במדיניות SCHED_FIFO:

$$\underbrace{A\ldots}_{\text{until finished}}\ ,\quad \underbrace{C\ldots}_{\text{until finished}}\ ,\quad \underbrace{D\ldots}_{\text{until finished}}\ ,\quad B\ldots$$

- אם כל התהליכים במדיניות SCHED_RR:

$$\underbrace{A\ldots}_{q}\ ,\quad \underbrace{C\ldots}_{q}\ ,\quad \underbrace{D\ldots}_{q}\ ,\quad \underbrace{A\ldots}_{q}\ ,\quad \underbrace{C\ldots}_{q}\ ,\quad \underbrace{D\ldots}_{q}$$

- ללא קשר למדיניות הזימון, תהליך B ירוץ רק כאשר שלושת התהליכים A,C,D יסתיימו ו/או לא יהיו מוכנים לריצה.

הוספת תהליכים לתור הריצה

- תהליכים חדשים ותהליכים שחזרו מהמתנה יכנסו לסוף התור המתאים לעדיפותם.

- שאלה: מה היתרונות בלהכניס לסוף התור לעומת תחילת התור?
  - אם המערכת לא עמוסה (כלומר מעט תהליכים) – זה כמובן לא משנה.
- תשובה:

1. הוספת תהליכים בסוף התור שומרת על הוגנות כי תהליכים שהגיעו קודם ירוצו קודם.
2. הוספת תהליכים בתחילת התור הייתה עלולה ליצור הרעבה אם תהליכים חדשים ממשיכים להגיע ותהליכים בסוף התור לא זוכים לרוץ.

חישוב ה-time slice

- כאמור, לכל תהליך במדיניות SCHED_RR מוקצב פרק זמן לשימוש במעבד - time slice או quantum.
- ה-time slice מוגדר ביחידות של פסיקות שעון.
  - בכל פסיקת שעון ערכו קטן ב-1.
  - כאשר הוא מגיע ל-0, התהליך סיים את הזמן שהוקצב לו.
- time slice מוקצה לתהליך במאקרו TASK_TIMESLICE.
  - החישוב עובר מיחידות של מילי-שניות ליחידות של מספר פסיקות שעון.
  - HZ = מספר פסיקות השעון בשניה.

```
#define RR_TIMESLICE    (100 * HZ / 1000)
```

זימון תהליכים רגילים בלינוקס

האבולוציה של זימון תהליכים בלינוקס

אלגוריתם הזימון של גרסה 2.4 – נלמד בהרצאה.

פועל בסיבוכיות ליניארית O(N) ולכן **לא סקלבילי**.

אלגוריתם הזימון של גרסה 2.6 – לא נלמד בקורס.

פועל בסיבוכיות קבועה O(1), אבל בפועל **איטי מאוד** בגלל חישובים מורכבים שמנסים לסווג בין תהליכים אינטראקטיביים לתהליכים חישוביים.

אלגוריתם הזימון החל מגרסה 2.6.23 – נלמד בתרגולים.

פועל בסיבוכיות לוגריתמית O(logN) וגם **מהיר בפועל**.

Completely Fair Scheduler (CFS)

- אלגוריתם הזימון של גרעין לינוקס החל מגרסה 2.6.23 הוא CFS.
- פותח כדי להשיג:
  - יעילות – האלגוריתם מבזבז מעט זמן על קבלת החלטות.
  - מדרגיות (scalability) – הביצועים מדרדרים בצורה מתונה יחסית כאשר מספר התהליכים גדל.

- במקום לנסות להקטין את זמן התגובה או זמן ההמתנה, CFS פשוט מנסה להיות הוגן ולתת לכל תהליך נתח שווה של זמן המעבד.
  - עד כדי עדיפויות: תהליכים בעדיפות גבוהה יותר יקבלו נתח גדול יותר.

- לצורך פעולתו, אלגוריתם CFS מגדיר מושג חדש:
  זמן ריצה וירטואלי.

זמן ריצה וירטואלי (vruntime)

- כאשר תהליך רץ, הוא צובר לעצמו זמן ריצה וירטואלי.
- באופן אידיאלי, זמן הריצה הווירטואלי שווה בין כל התהליכים בכל נקודת זמן.
- אבל באופן מעשי, רק תהליך אחד יכול לרוץ על המעבד בכל רגע נתון, ולכן יהיו הפרשים בין תהליכים שונים.

- כאשר CFS מזמן תהליך לריצה הוא יבחר את התהליך בעל זמן הריצה הווירטואלי הנמוך ביותר (כדי להיות הוגן).

- האלגוריתם שומר בכל רגע את המקסימום והמינימום של זמן הריצה הווירטואלי על-פני כל התהליכים המוכנים לריצה – מיד נבין למה.

אופן פעולת CFS

- האלגוריתם קובע טווח זמן שבמהלכו הוא ינסה להריץ את כל התהליכים (בדומה ל- epoch של אלגוריתם RR):

$$sched\_latency = 48\ ms$$

- האלגוריתם מקצה לכל תהליך פיסת זמן (בדומה לקוונטום של אלגוריתם RR) שבו הוא מקבל את המעבד. אם יש N תהליכים במערכת וכולם באותה עדיפות, הקוונטום של כל תהליך יהיה:

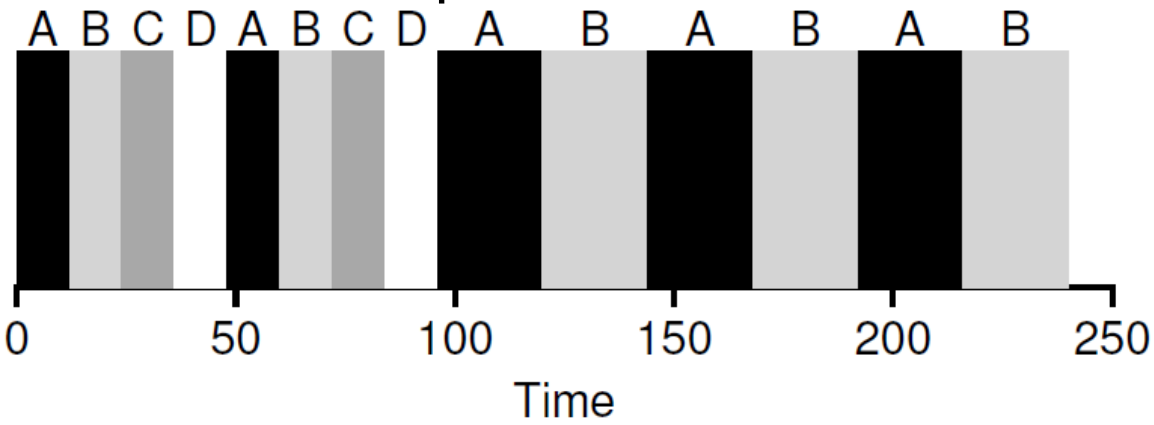
$$Qi = sched\_latency / N$$

- כאשר תהליך מסיים את הקוונטום שלו, האלגוריתם בוחר לריצה את התהליך בעל זמן הריצה הוירטואלי הנמוך ביותר בעץ.

דוגמת הרצה

- נניח כי במערכת יש ארבעה תהליכים A,B,C,D.
- אז הקוונטום של כל תהליך יהיה 12ms.
- כעת נניח כי לאחר שני epochs התהליכים C,D מסתיימים.
- אז התהליכים הנותרים A,B ממשיכים לרוץ עם קוונטום של 24ms.

- הזימון של CFS נראה כמו RR עם קוונטום דינמי:



מה קורה במערכת עמוסה?

- אם מספר התהליכים N במערכת גבוה, המערכת עלולה לסבול מהחלפות הקשר תכופות ופגיעה בביצועים.
- לכן מוגדר גם זמן מינימום על הקוונטום:

$$Qi \geq min\_granularity = 6\ ms$$

- לדוגמה, אם יש 10 תהליכים במערכת, אז הקוונטום של כל אחד אמור להיות:

$$Qi = 48\ ms\ /\ 10 = 4.8\ ms$$

- בפועל, כל תהליך יקבל 6 ms ולכן משך הסיבוב שבו כל התהליכים ירוצו יהיה:

$$epoch = 60\ ms \geq sched\_latency$$

הבדלים בין CFS ו–RR

CFS

1. כאשר תהליך מסיים את הקוונטום שלו, CFS בוחר לריצה את התהליך בעל זמן הריצה הוירטואלי הנמוך ביותר בעץ.
2. הקוונטום משתנה באופן דינמי בהתאם למספר התהליכים במערכת.

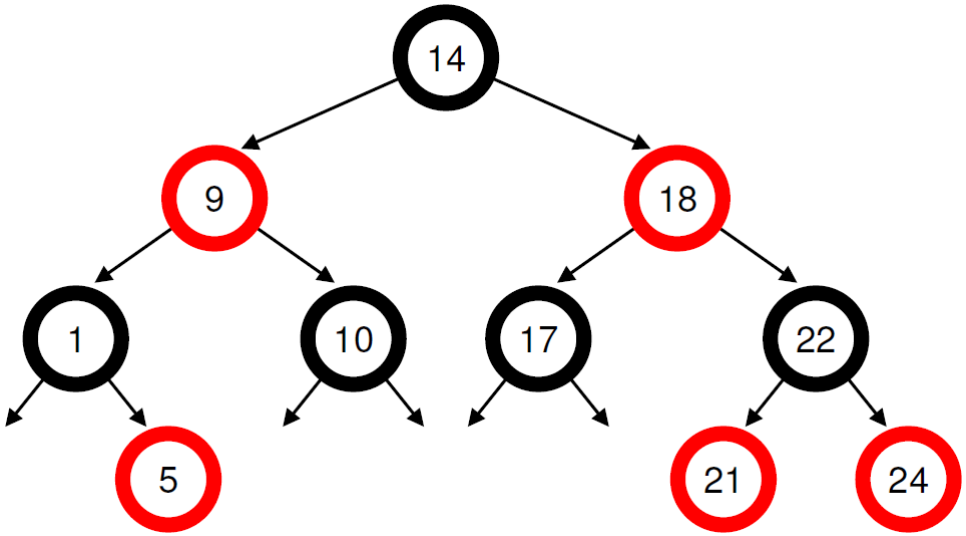
RR

1. כאשר תהליך מסיים את הקוונטום שלו, RR בוחר לריצה את התהליך הבא ברשימה המעגלית.
2. הקוונטום סטטי ואינו תלוי במספר התהליכים במערכת.

מבנה הנתונים של CFS

- עץ אדום–שחור – עץ חיפוש בינארי מאוזן עם סיבוכיות חיפוש, הכנסה, ומחיקה לוגריתמיות.
  - O(logN) כאשר N הוא מספר התהליכים.
- מפתח החיפוש בעץ הוא זמן ריצה הווירטואלי (vruntime).
- תהליכים חדשים מאותחלים עם vruntime = max_vruntime .

למה?



יציאה וחזרה מהמתנה

- רק תהליכים מוכנים לריצה נשמרים בעץ.
- תהליכים שמבצעים I/O יוצאים מהעץ ועוברים לתור המתנה.

- תרחיש בעייתי:
  - שני תהליכים A,B רצים זה לצד זה.
  - תהליך B יוצא להמתנה ארוכה של 10 שניות.
  - כאשר B מתעורר, הוא נכנס לעץ עם vruntime קטן ב-10 שניות מזה של A.
  - לפי CFS, תהליך B יהיה זה שירוץ ב-10 השניות הבאות.
  - ← הרעבה של תהליך A.

- כדי למנוע את התרחיש הבעייתי הנ"ל, CFS מוסיף תהליכים שחזרו מהמתנה ארוכה עם vruntime = min_vruntime .

עדיפויות

- CFS מאפשר למשתמש להגדיר עדיפויות לתהליכים וכך לחלק את זמן המעבד בצורה שונה בין התהליכים.
- העדיפות של התהליך מיוצגת ע"י הערך $-20 \le nice \le +19$.
  - ברירת המחדל היא nice=0 .
  - תהליך "נחמד" יותר יהיה בעדיפות נמוכה יותר.
- לכל עדיפות יש משקל:

```
static const int 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,
};
```

קצת נוסחאות

- נניח שיש במערכת n תהליכים עם עדיפויות: P1, P2, …, Pn ומשקלים: W1, W2, …, Wn .
- נניח כי W0 הוא המשקל המתאים לעדיפות nice=0.

- אז זמן הריצה הווירטואלי של התהליך ה-i מתקדם לפי:

$$VRi \mathrel{+}= (W0 \ / \ Wi) \bullet \Delta T$$

- כאשר ΔT הוא זמן הריצה לפי שעון אמיתי.
  - זמן הריצה הווירטואלי זהה לזמן הריצה האמיתי עבור ברירת המחדל nice=0.

- ניתן להוכיח כי הקוונטום של התהליך ה-i הוא:

$$Qi = (Wi \ / \ \Sigma Wi) \bullet sched\_latency$$

דוגמת חישוב

- נניח כי יש שני תהליכים:
- P1 עם ערך nice = -5 ← משקל W1 = 3121 .
- P2 עם ערך nice = 0 ← משקל W2 = 1024 .

- נקבל כי תהליך P2 מתקדם (בערך) פי 3 יותר מהר מתהליך P1:

$$VR1 \mathrel{+}= 1/3 \bullet \Delta T$$
$$VR2 \mathrel{+}= \Delta T$$

- ואכן:

$$Q1 = \tfrac{3}{4} \bullet sched\_latency$$
$$Q2 = \tfrac{1}{4} \bullet sched\_latency$$