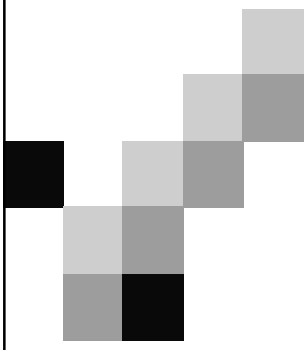
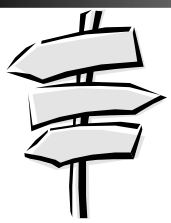


זימון תהליכים ב-Linux

ארז חדד

(מבוסס על חומר מהקורס "מערכות הפעלה" בטכניון)



תוכן

- אלגוריתם זימון התהליכים ב-Linux
  - כיצד נבחר תהליך לריצה במעבד בכל זמן נתון?

הערה מקדימה

- אלגוריתם הזימון שמוצג בתרגול זה הוא האלגוריתם החדש של Linux – גרסאות גרעין 2.5.X ומעלה
  - אלגוריתם זה אינו מתועד כיאות ובפרט אינו מופיע בספר Understanding the Linux Kernel
  - לקבלת מידע באינטרנט על האלגוריתם החדש, חפשו מידע על על קבצי קוד הגרעין העוסקים בזימון בגרסאות בטא של הגרעין העתידי 2.5.X
    - הקבצים העיקריים: include/linux/sched.h ו-kernel/sched.c
    - ניתן לחפש גם לפי שם מחבר האלגוריתם החדש: Ingo Molnar

זימון תהליכים ב-Linux (1)

- Linux מאפשרת למספר תהליכים לרוץ "בו זמנית" על המעבד באמצעות חלוקת זמן המעבד ביניהם (time sharing)
  - העברת המעבד מתהליך אחד לאחר באמצעות החלפת הקשר
- עבור תהליכים "רגילים", חלוקת הזמן נאכפת באמצעות מערכת ההפעלה, גם אם תהליכים לא מוותרים מרצונם על המעבד
  - Preemptive multitasking
- מדיניות זימון התהליכים ה"רגילים" ב-Linux מסומנת SCHED_OTHER
  - ישנן מדינויות זימון נוספות כפי שנראה מיד
  - לכל תהליך מוגדר מהי מדיניות הזימון לה הוא כפוף
  - בהמשך נתמקד במדיניות זו בלבד

זימון תהליכים ב-Linux (2)

- Linux תומכת, בנוסף לתהליכים "רגילים", בתהליכי זמן-אמת (real-time)
  - תהליכים הצריכים להגיב על אירועים שונים במערכת בזמן קצר ואחיד ככל האפשר
  - דוגמאות: יישומי מולטימדיה, בקרה תעשייתית וכו'
- תהליכי זמן-אמת מועדפים לריצה על-פני תהליכים רגילים
  - תהליכים רגילים אינם זוכים לרוץ אם יש תהליכי זמן-אמת מוכנים לריצה (מצב TASK_RUNNING)
- עבור תהליכי זמן-אמת קיימים שני סוגים נוספים של מדיניות זימון:
  - SCHED_FIFO: שיתוף פעולה (non-preemptive) עם עדיפויות – המעבד מתפנה רק כאשר התהליך מחליט לוותר עליו או כאשר תהליך עדיף יותר מוכן לריצה. המעבד מוקצה לתור התהליכים העדיפים ביותר המוכנים לריצה
  - SCHED_RR: חלוקת זמן שווה (Round Robin) – זמן המעבד מחולק בצורה שווה בין כל התהליכים העדיפים ביותר המוכנים לריצה

אלגוריתם הזימון (SCHED_OTHER)

- זמן המעבד מחולק ל"תקופות" (epochs)
- לכל תהליך מוקצב פרק זמן לשימוש במעבד – time slice
  - בסיום ה-time slice נבחר מתוך ה-runqueue תהליך לריצה במעבד
  - התהליך הנבחר לריצה הוא תהליך מוכן לריצה (TASK_RUNNING) בעל העדיפות הגבוהה ביותר שנותר לו זמן ריצה מתוך ה-time slice שלו
- בכל תקופה, כל התהליכים המוכנים לריצה מזומנים למעבד עד שכל תהליך מכלה את ה-time slice שלו או יוצא להמתנה
- תקופה מסתיימת כאשר כל תהליך שעדיין מוכן לריצה סיים את ה-time slice שלו
- לאחר סיום תקופה מתחילה תקופה חדשה בה לכל תהליך המוכן לריצה מוקצה time slice חדש

מימוש אלגוריתם הזימון

- אלגוריתם הזימון ממומש בגרעין Linux ע"י רכיבי הקוד הבאים (בקובץ kernel/sched.c)
  - זַמָּן התהליכים (scheduler) שאחראי על בחירת התהליך הבא לריצה במעבד ועל החלפת תקופות
    - הפונקציה ()schedule
  - מנגנון עדכון דינמי (מדי פעימת שעון) של נתוני זימון של התהליך הנוכחי שרץ במעבד (עדיפות, time_slice, וכו') והפעלת הזַמָּן לפי הצורך
    - הפונקציה ()scheduler_tick
  - כמו כן, כפי שנראה בהמשך, ביציאה ובחזרה מהמתנה:
    - מעודכנים נתוני הזימון של התהליך שיוצא/חוזר
    - מופעל הזַמָּן כדי לבחור מחדש תהליך לריצה במעבד
    - הפונקציות ()(de)activate_task

מדידת זמן

- הזמן נמדד במחשב באמצעות פעימות שעון בתדר מוגדר מראש
  - השעון הוא התקן חומרה חיצוני למעבד
  - כל פעימת שעון יוצרת פסיקת חומרה
- ב-Linux השעון מתוכנת לתדר של HZ=100Hz
- הזמן מאז הפעלת המחשב (בפעימות שעון) נמדד ב-Linux בתוך מונה הקרוי jiffies

עדיפות תהליך (1)

- בחירת התהליך הבא לזימון למעבד מבוססת על עדיפות התהליך (process priority)
- עדיפות תהליך נקבעת באופן דינמי במהלך חייו
  - ערך בסיסי קבוע - עדיפות סטטית
    - תהליך בן יורש את העדיפות הסטטית מאביו
    - ניתן לשנות עדיפות סטטית על-ידי API (קריאות כדוגמת nice())
  - שינוי דינמי לערך הבסיס בהתאם לפעילות התהליך
    - הרעיון: "לפצות" תהליך שצריך להמתין הרבה בטווח הבינוני/ארוך על-ידי הגברת עדיפותו לריצה בטווח הקצר
    - באופן כללי, תהליכים שהם עתירי ק/פ (IO-BOUND) מועדפים על-פני תהליכים שהם עתירי חישוב (CPU-BOUND) בעלי אותה עדיפות סטטית

עדיפות תהליך (2)

- העדיפות של תהליך נקבעת כדלהלן:
  - דרגות העדיפות הן 0..139 (MAX_PRIO-1)
  - עדיפויות 0..99 (MAX_RT_PRIO-1) מיועדות לתהליכי זמן-אמת
  - עדיפויות 100..139 מיועדות לתהליכים רגילים
- ערך עדיפות מספרי גבוה מציין עדיפות נמוכה, וההפך
- תהליך רגיל מתחיל בדרך-כלל עם עדיפות 120
  - הכוונה לעדיפות סטטית 120
- ההפרש בין העדיפות הסטטית של תהליך ל-120 מוגדר כ-niceness (ערך nice) של התהליך
  - עבור תהליך רגיל, ערך ה-nice נע בין 20- ל-19+
  - החישוב מקובע במאקרו TASK_NICE:

```
#define TASK_NICE(p)  ((p)->static_prio – MAX_RT_PRIO – 20)
```

מבט מקרוב על ה-runqueue (1)

- ה-runqueue הוא מבנה נתונים המאחסן את כל מתארי התהליכים (process descriptors) של התהליכים הרצים או מוכנים לריצה על מעבד מסוים ("הטווח הקצר")
- כל runqueue (על כל מעבד) מכיל את הנתונים הבאים:
  - nr_running – מספר התהליכים ב-runqueue (לא כולל את ה-swapper)
  - curr – מצביע למתאר התהליך שרץ כרגע
  - idle – מצביע למתאר תהליך ה-swapper

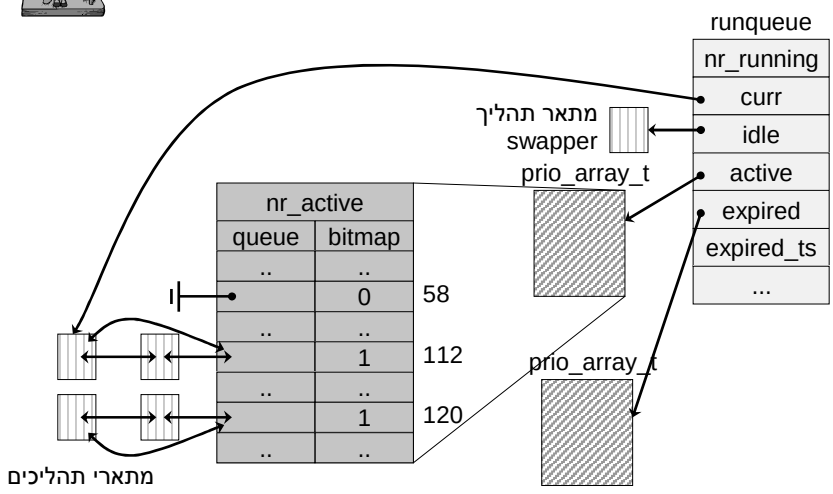
מבט מקרוב על ה-runqueue (2)

  - active – מצביע למערך תורי העדיפויות של התהליכים הפעילים, כלומר תהליכים מוכנים לריצה (נמצאים במצב TASK_RUNNING) שנותר להם זמן ריצה במסגרת התקופה הנוכחית
    - תהליך החוזר מהמתנה בטווח הבינוני משובץ לריצה בתורי מערך זה לפי עדיפותו בפונקציה activate_task()
  - expired – מצביע למערך תורי העדיפויות של התהליכים המוכנים לריצה שכילו את ה-time slice שלהם בתקופה הנוכחית
  - expired_timestamp – מתי עבר התהליך הראשון מ-active ל-expired בתקופה הנוכחית

מבט מקרוב על ה-runqueue (3)

- מערך תורי עדיפויות (struct prio_array) מכיל את הנתונים הבאים:
  - nr_active – מספר התהליכים המצויים במערך זה
  - []bitmap – וקטור ביטים בגודל מספר דרגות העדיפות (140 ביטים)
    - ביט i דלוק (1) אם יש תהליכים בתור המתאים לעדיפות i
    - התהליך הבא לזימון הוא התהליך הראשון בתור העדיפות הנמוך ביותר ב-active שהביט שלו דלוק – חישוב ב-O(1)
  - []queue – מערך התורים עצמו, עם כניסה לכל דרגת עדיפות (140 כניסות). כל כניסה מכילה ראש תור מסוג list_t.

מבט מקרוב על ה-runqueue (4)

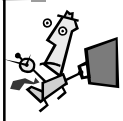


נתוני תהליך המשמשים לזימון (1)

- מתאר תהליך מכיל את הנתונים הבאים המשמשים בזימון:
  - policy – מהי מדיניות הזימון של התהליך
  - need_resched – דגל: האם התהליך אמור לוותר על המעבד
    - המאקרו set_tsk_need_resched מדליק את הדגל
    - המאקרו clear_tsk_need_resched מכבה את הדגל

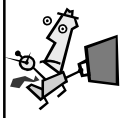
נתוני תהליך המשמשים לזימון (2)

- prio – עדיפות התהליך
- static_prio – העדיפות הסטטית של התהליך – נראה בהמשך
- sleep_timestamp – מתי לאחרונה בוצעה החלפת הקשר מהתהליך (כלומר התהליך הנ"ל הוא זה שוויתר על המעבד)
- sleep_avg – זמן המתנה "ממוצע" של התהליך
  - משמש לאפיון התנהגות התהליך – פרטים בהמשך
- time_slice – כמות הזמן (בפעימות שעון) הנותרת לתהליך לריצה במהלך התקופה (epoch) הנוכחית



זַמָּן התהליכים – ה-scheduler (1)

- זַמָּן התהליכים הוא רכיב תוכנה בגרעין Linux שאחראי על זימון התהליך הבא למעבד
  - קוד הזמן – הפונקציה ()schedule
  - הזמן בוחר את התהליך הבא ואז מפעיל את פונקצית החלפת ההקשר ()context_switch – פירוט בהמשך
- הזמן מופעל במספר מקרים אופייניים:
  - בתגובה על פסיקת שעון – לאחר שתהליך כילה את ה-time slice שלו (בעקבות השגרה ()scheduler_tick)
  - בטיפול בקריאת מערכת הגורמת לתהליך הקורא לעבור להמתנה, כך שהמעבד מתפנה להריץ תהליך אחר (שגרות כדוגמת ()sleep_on)
  - בחזרה של תהליך מהמתנה – יש לשבץ את התהליך ב-runqueue ולבדוק אם כתוצאה מכך יש לבצע החלפת הקשר (שגרות כמו ()wake_up)
  - כאשר תהליך מחליט לוותר עם המעבד מרצונו בקריאת מערכת כגון ()sched_yield



זַמָּן התהליכים – ה-scheduler (2)

- זמן התהליכים מופעל כאשר פסיקות החומרה חסומות
  - כך נמנעת הפעלה מקבילה נוספת של הזמן בעקבות פסיקת שעון
- הפעלת הזמן באופן ישיר (direct invocation) מבוצעת על-ידי קריאה ל-()schedule
  - למשל מתוך פונקציות המתנה
- הזמן מופעל גם באופן עקיף (lazy invocation) על-ידי הדלקת דגל need_resched במתאר התהליך לפני חזרה מ-kernel mode ל-user mode
  - בכל חזרה מפסיקה הקוד בודק אם הדגל הנ"ל דלוק ואם כן מפעיל את ()schedule
  - דוגמה אופיינית: במהלך טיפול בפסיקת השעון, אם נגמר ה-time slice של התהליך, השגרה ()scheduler_tick מדליקה את הדגל need_resched במתאר התהליך

הפונקציה schedule() (1)

```
prev = current;
rq = this_rq();
..
spin_lock_irq(&rq->lock);
switch(prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        deactivate_task(prev, rq);
    case TASK_RUNNING:
        ;
}
```

prev מצביע למתאר התהליך הנוכחי (שמוותר על המעבד)
rq מכיל את ה-runqueue של המעבד הנוכחי

חסימת הפסיקות

רק תהליך שעבר למצב המתנה מוצא מתוך ה-runqueue למעט מצב של הפרעה בהמתנה

הפונקציה schedule() (2)

```
if (!rq->nr_running) {
    next = rq->idle;
    rq->expired_timestamp = 0;
    goto switch_tasks;
}
```

אם אין יותר תהליכים לזימון מתוך ה-runqueue, יזומן תהליך ה-swapper

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
}
```

אם לא נותרו תהליכים ב-active array, מחליפים בין ה-active וה-expired (סיום תקופה והתחלת תקופה חדשה)

הפונקציה schedule() (3)

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);

switch_tasks:
clear_tsk_need_resched(prev);
if (prev != next) {
    ..
    rq->curr = next;
    prev = context_switch(prev, next);
    ..
    spin_unlock_irq(&rq->lock);
}
else
    spin_unlock_irq(&rq->lock);
```

התהליך הבא לזימון למעבד (next) הוא התהליך הראשון בתור ב-active array בעל העדיפות הגבוהה ביותר

ביצוע החלפת ההקשר ע"י קריאה ל-()context_switch

שאלה: מי מבצע את אפשור הפסיקות מחדש לאחר הקריאה ל-()context_switch?

תשובה: התהליך next

עדכון דינמי של נתוני זימון

- הפונקציה scheduler_tick() – מופעלת מדי פעימת שעון
- מעדכנת את נתוני הזימון של התהליך הנוכחי שרץ במעבד:
  - אפיון התנהגות התהליך באמצעות איסוף סטטיסטיקה
    - "זמן המתנה ממוצע"
  - הקטנת ה-time_slice, ובסיומו:
    - חישוב מחדש של העדיפות הדינמית בהתאם לאופי התהליך
    - הקצאת time_slice חדש
    - בדיקת תהליך אינטראקטיבי והעברת התהליך בהתאם: ל-expired או שוב ל-active
    - בקשה להפעלת הזַמָן – need_resched

אפיון התנהגות תהליך (1)

- Linux מודדת את "זמן ההמתנה הממוצע" של תהליך בשדה sleep_avg במתאר התהליך
  - "זמן המתנה ממוצע" = סך כל זמן ההמתנה בטווח הבינוני פחות סך כל זמן הריצה
- בכל פעם שתהליך מוותר על המעבד, ערך השעון נשמר (פונקצית schedule()):

```
p->sleep_timestamp = jiffies;
```

- כאשר תהליך חוזר מהמתנה מוסף זמן ההמתנה לחשבון (פונקציה activate_task()) עד לגודל מקסימלי MAX_SLEEP_AVG

```
#define MAX_SLEEP_AVG (2*HZ)
sleep_time = jiffies – p->sleep_timestamp;
p->sleep_avg += sleep_time;
if (p->sleep_avg > MAX_SLEEP_AVG)
   p->sleep_avg = MAX_SLEEP_AVG;
```


אפיון התנהגות תהליך (2)

- כל פעימת שעון בה התהליך רץ מורידה מהממוצע (הפונקציה sheduler_tick()) עד למינימום 0

```
if (p->sleep_avg)
   p->sleep_avg--;
```

- על-פי שיטת חישוב זו
  - תהליך עתיר חישוב צפוי להגיע ל"זמן המתנה ממוצע" נמוך
  - תהליך עתיר ק/פ צפוי להגיע ל"זמן המתנה ממוצע" גבוה

חישוב דינמי של עדיפות תהליך (1)

- Linux מעדכנת את העדיפות של כל תהליך "רגיל" באופן דינמי בהתאם ל"זמן ההמתנה הממוצע" של התהליך – sleep_avg
  - עדיפויות תהליכי זמן-אמת מקובעות לערך הסטטי
- חישוב העדיפות הדינמית מתבצע בפונקציה effective_prio() לפי האלגוריתם הבא:

$$bonus = 25\% \times 40 \times \left(\frac{sleep\_avg}{MAX\_SLEEP\_AVG} - \frac{1}{2} \right) \qquad -5 \le bonus \le 5$$

```
prio = static_prio – bonus
if (prio < MAX_RT_PRIO)
  prio = MAX_RT_PRIO;
if (prio > MAX_PRIO - 1)
  prio = MAX_PRIO – 1;
```


חישוב דינמי של עדיפות תהליך (2)

- הגבלת גודל ה-bonus ל-5 נועדה למנוע מצב שבו יתבצע היפוך עדיפויות בין תהליכים שעדיפויותיהם הבסיסיות (הסטטיות) רחוקות זו מזו
  - למשל, שתהליך בעל nice 19 (עדיפות 139) יהפוך לעדיף יותר מתהליך בעל nice 0 (עדיפות 120)
- שיטת חישוב זו משפרת את העדיפות של תהליך ש"ממתין הרבה" (צפוי לתהליכים עתירי ק/פ) ומרעה את העדיפות של תהליך ש"ממתין מעט" (צפוי לתהליכים עתירי חישוב)
  - הערך של מחצית MAX_SLEEP_AVG, כלומר HZ פעימות או שניה אחת, נקבע בתור הסף בין "המתנה מרובה" ל"המתנה מועטת"

חישוב דינמי של עדיפות תהליך (3)

- חישוב מחדש של העדיפות הדינמית מתבצע רק כאשר תהליך מכלה time slice או כאשר תהליך חוזר מהמתנה
  - תהליך המקבל עדיפות גבוהה בעקבות המתנה יכול לרוץ את מלוא הזמן המוקצב לו בעדיפות שקיבל

הקצאת time slice לתהליך (1)

- time slice מוקצה לתהליך במאקרו TASK_TIMESLICE:

```
#define MIN_TIMESLICE   (10 * HZ / 1000)              /* 10 msec */
#define MAX_TIMESLICE  (300 * HZ / 1000) /* 300 msec */
#define TASK_TIMESLICE(p) \
    MIN_TIMESLICE + (MAX_TIMESLICE – MIN_TIMESLICE) * (MAX_PRIO – 1 – (p)->static_prio)/39
```

- כל תהליך מקבל לפחות 10 msec זמן ריצה תלוי ליניארית בעדיפות הסטטית אך לא יותר מ-300 msec
  - תהליך בעדיפות "רגילה" (120) מקבל זמן טיפוסי 150 msec
- חישוב time slice מחדש מבוצע אך ורק עם סיום ה-time slice הנוכחי כפי שנראה בפונקציה scheduler_tick() בהמשך
  - כלומר, אם תהליך יוצא להמתנה בטווח הבינוני וחוזר לטווח הקצר, עליו לסיים תחילה את שארית ה-time slice שלו לפני שיקבל הקצאה חדשה

הקצאת time slice לתהליך (2)

- ביצירת תהליך חדש, תהליך האב מאבד מחצית מה-time slice שלו לטובת תהליך הבן
- המטרה: למנוע מצב בו תהליך יכול להרוויח זמן מעבד בתוך אותה תקופה, למשל באופן הבא:
  - לפני סיום ה-time slice, התהליך מייצר תהליך בן שממשיך להריץ את הקוד למשך time slice נוסף באותה תקופה
  - לפני שה-time slice של הבן מסתיים, הבן מייצר נכד, וכן הלאה..
- מימוש המנגנון בפונקציה do_fork() שמבצעת את fork()
  - קובץ גרעין kernel/fork.c

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```


תהליכים אינטראקטיביים (1)

- תהליכים אינטראקטיביים מקבלים ב-Linux העדפה מיוחדת: זכות לרוץ time slices נוספים בתקופה הנוכחית
  - תהליכים אינטראקטיביים הם תהליכים הממתינים לקלט מהמשתמש, כדוגמת shell, תוכנה חלונאית וכו' – המתנות ארוכות במיוחד
  - המטרה בהעדפת תהליכים אינטראקטיביים היא לגרום להם להגיב בצורה מהירה על פעולות המשתמש – תחושה נוחה למשתמש בעבודה מול יישומים
- העדפת תהליכים אינטראקטיביים ללא הגבלה תגרום לכך שתהליכים לא-אינטראקטיביים שסיימו את ה-time slice בתקופה הנוכחית "יורעבו", כלומר לא יינתן להם לרוץ ולהתקדם

תהליכים אינטראקטיביים (2)

- כדי למנוע את תופעת ההרעבה, מוגדר "סף הרעבה" על זמן ההמתנה של התהליכים ב-expired:

```
#define EXPIRED_STARVING(rq) \
    ((rq)->expired_timestamp && \
    (jiffies – (rq)->expired_timestamp >= \
        STARVATION_LIMIT * ((rq)->nr_running + 1))
```

- כאשר התהליכים ב-expired מורעבים מעבר לסף, מופסקת הענקת time slices נוספים לתהליכים אינטראקטיביים בתקופה הנוכחית
- הסף המוגדר פרופורציוני למספר התהליכים ב-runqueue, כך שככל שיש יותר תהליכים מוכנים לריצה, הסף גבוה יותר
  - ככל שהעומס במערכת גבוה יותר, תהליכים אינטראקטיביים מקבלים יותר העדפה על מנת לשמור על מהירות תגובה נאותה לפעולות המשתמש

תהליכים אינטראקטיביים (3)

- סיווג תהליך כאינטראקטיבי מבוצע במאקרו TASK_INETRACTIVE:

```
#define TASK_INTERACTIVE(p) \
    ((p)->prio <= (p)->static_prio – DELTA(p))
```

כאשר DELTA(p) ניתנת להגדרה כדלקמן:

$$DELTA(p) = 25\% \times 40 \times \frac{1}{2} \times \frac{TASK\_NICE(p)}{20} + 2 = 5 \times \frac{TASK\_NICE(p)}{20} + 2$$

- תהליך מסווג כאינטראקטיבי אם העדיפות (הדינמית) שלו "חורגת" מהעדיפות הסטטית שלו
  - תוצאה של זמן המתנה ממוצע גבוה המקנה בונוס בחישוב העדיפות הדינמית של התהליך

תהליכים אינטראקטיביים (4)

- ככל שעולה העדיפות הסטטית של התהליך, "קל יותר" לתהליך להסתווג כאינטראקטיבי
  - "קל יותר": התהליך צריך לצבור זמן המתנה ממוצע נמוך יותר על-מנת להיחשב כאינטראקטיבי
  - לדוגמה: תהליך עם nice -20 יכול להיות "חזיר" בצריכת המעבד ולהגיע לעדיפות דינמית של בונוס שלילי 3- ועדיין להסתווג כאינטראקטיבי. לעומת זאת, תהליך עם nice 19 לא יכול כלל להסתווג כאינטראקטיבי כי נדרש בונוס של 7+ והבונוס המקסימלי הוא 5+
- הרעיון: תהליכים בעלי עדיפות סטטית נמוכה הם בדרך-כלל תהליכי אצווה (batch) שאינם כוללים אינטראקציה עם המשתמש
  - דוגמאות: חישובים מדעיים, גיבוי אוטומטי, וכו'

הפונקציה scheduler_tick() (1)

- פונקציה זו מופעלת בכל פעימת שעון ומעדכנת נתוני זימון של תהליכים
  - גם פונקציה זו מופעלת כשהפסיקות חסומות כדי למנוע שיבוש נתונים ע"י הפעלת הפונקציה במקביל
  - הפונקציה תגרום להפעלת schedule() אם התהליך הנוכחי סיים את ה-time slice שלו
  - הפונקציה משתמשת בפונקציות enqueue_task() ו-dequeue_task() להוצאה והכנסה של תהליך לאחד ממערכי התורים (active או expired)

```
task_t *p = current;
runqueue_t *rq = this_rq();
..
if (p->sleep_avg)
    p->sleep_avg--;
```

עדכון ה-sleep_avg של התהליך הנוכחי

הפונקציה scheduler_tick() (2)

```
if (!--p->time_slice) {
    dequeue_task(p, rq->active);
    set_tsk_need_resched(p);
    p->prio = effective_prio(p);
    p->time_slice = TASK_TIMESLICE(p);

    if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
        if (!rq->expired_timestamp)
            rq->expired_timestamp = jiffies;
        enqueue_task(p, rq->expired);
    } else
        enqueue_task(p, rq->active);
```

אם התהליך הנוכחי כילה את ה-time slice של עצמו, הוא מוצא מה-active, העדיפות הדינמית וה-time slice הבא שלו מחושבים מחדש, ויבוצע לו schedule() בהמשך

כאשר התהליך הראשון בתקופה הנוכחית עובר ל-expired, מעודכן expired_timestamp

תהליך אינטראקטיבי מוחזר ל-active לרוץ time slice נוסף בתקופה הנוכחית כאשר אין הרעבה