מערכות הפעלה - הרצאה 8

שרון מלטר, אתגר 17

26 ביולי 2024

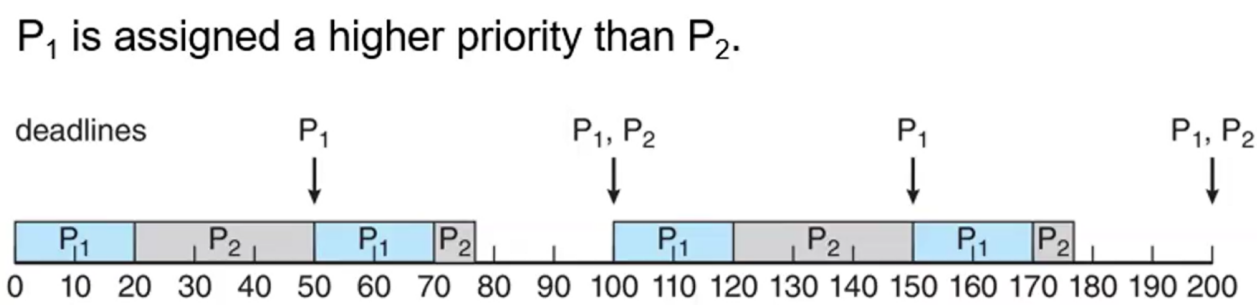
1 Chapter 5 Cont

1.1 Hard Real − Time Scheduling

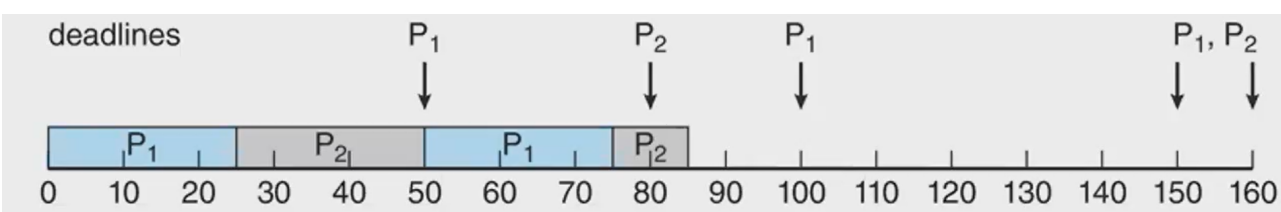
נמשיך עם כיצד $scheduler$ יכול ליישם מדיניות שמספקת את הדרישות של מערכת $hard\ real-time$.
בשיעור הקודם למדנו על $rates$, שהינם $1/p$ כאשר p הוא $period$ בו ניתן לעשות משימה כלשהי. הם מאפשרים להעניק עדיפויות למשימות ע"פ הגישה הבאה;

1.1.1 Rate Monotonic Scheduling

העדיפות של כל משימה היא ה־ $rate$ של המחזור שלה. להלן דוגמה; (המחזור של p_1 הוא 50 ושל p_2 הוא 100.
כלומר, p_1 צריך להתבצע פעם ב־05 יחידות זמן ו־ p_2 פעם ב־ 100 יחידות זמן) עם זאת, לא תמיד ניתן לסיים

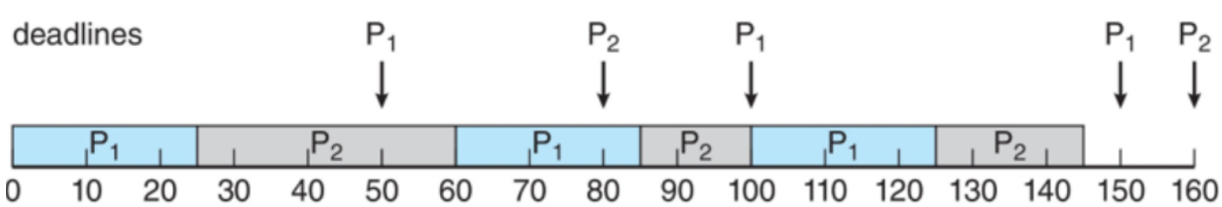


את כל המשימות בזמן. למשל, עם המחזור של p_2 הינו 80 אזי לא נוכל לסיים את שתי המשימות לכל $scheduling$;



1.1.2 Earliest Deadline First Scheduling (EDF)

העדיפות של משימות ניתנת לפי הדדליינים שלהם. כלל שהדדליין מוקדם יותר, כך קדמה העדיפות. לדוגמה;



1.1.3 *Proportional Share Scheduling*

ישנם T "*shares*" (משאבים של המעבד) לכל התהליכים במערכת. כל אפליקציה מקבלת $N < T$ מהם.
למדנו מספר אלגוריתמים שונים ל־ *scheduling*, אך איך בוחרים ביניהם?

1.2 הערכת אלגוריתם *Scheduling*

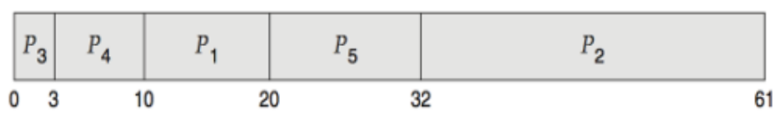
אז איך מעריכים את טיבו של אלגוריתם *scheduling*?

1.2.1 הערכה דטרמיניסטית

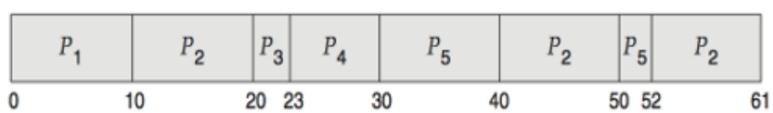
לכל אלגוריתם, נחשב ממוצע ה־ *waiting time*. כך נוכל למצוא את המינימום מבין הזמנים ולקבל את האלגוריתם הטוב ביותר. השיטה קלה לביצוע, אך דורשת קלט מסוים ורלוונטית רק אליו.
דוגמה:

- FCS is 28ms:

- Non-preemptive SFJ is 13ms:



- RR is 23ms:



1.2.2 *Queueing Models*

מנתחים את זמני ההגעה של תהליכים ל־ *ready queue* ואת ה־ *CPU* ו־ *I/O bursts* שלהם. לרוב נתונים אלה אקספוננציאליים ומתוארים על ידי התוחרת שלהם. בעזרתם מנתחים את ממוצעי ניצול המשאבים וה־ *time waiting* שיתקבלו עבור שימוש בכל אלגוריתם.
המערכת עצמה מתוארת כרשת של שרתים, כך שלכל אחד מהם תור של תהליכים המחכים לרוץ. מהכמות התהליכים שאמורים להגיע מחשבים את יעילות המערכת, אורך התור הממוצע וה־ *waiting time* הממוצע וכו'.

1.2.3 *Little's Formula*

המשך למודל התורים.

- n ־ אורך התור הממוצע.
- W ־ ה־ *waiting time* הממוצע בתור.
- λ ־ קצב ההגעה הממוצע של תהליכים לתור.

החוק של ליטל אומר כי במצב יציב, מספר התהליכים שעוזבים תור חייב להיות שווה למספר התהליכים שנכנסים אליו. כלומר;

$$n = \lambda \times W$$

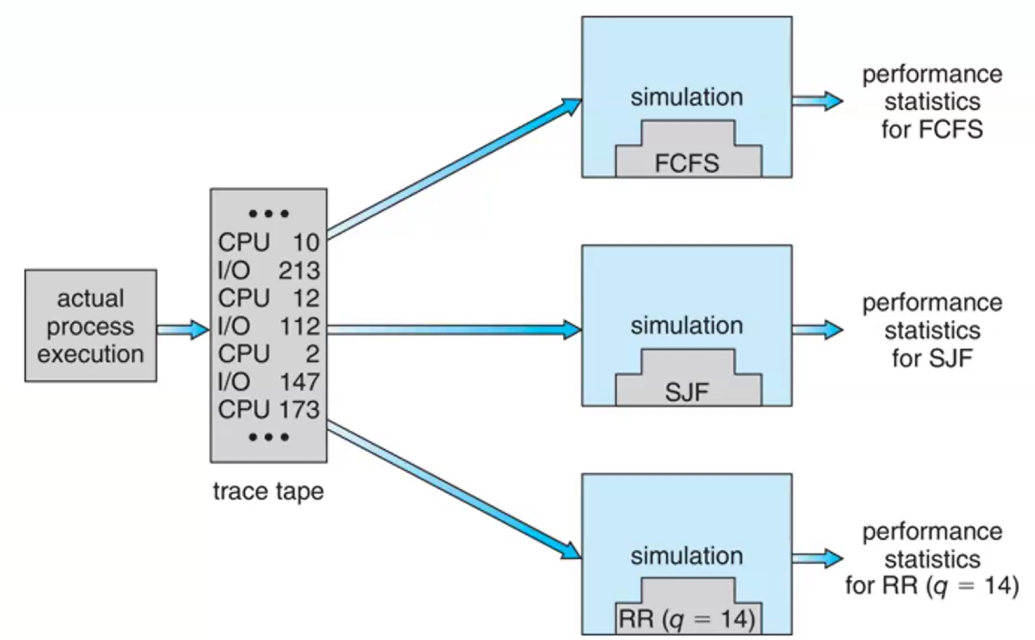
נוסחה זו נכונה לכל אלגוריתם *scheduling* שנבחר.

1.2.4 סימולציות

כפי שראינו, גם בתיאור המערכת וגם בדרישות, מודל התורים הינו מוגבל. סימולציות נוטות להיות יותר מדויקות. נבנה מודל וירטואלי של מערכת ההפעלה, נריץ אותה עבור דאטה (זמני הגעה של תהליכים, ה־ CPU ו־ $bursts$ I/O שלהם) וכך נקבל סטטיסטיקה על יעילות אלגוריתם ה־ $scheduling$.
ישנן שלוש דרכים ליצור דאטה עבור הסימולציה־

- הגדרת מספרים רנדומליים לפי סטטיסטיקות.
- על ידי התפלגויות שמתקבלות באופן מתמטי או אמפירי.
- $Trace\ tapes$ של סדרות אירועים אמיתיות ממערכות אמיתיות.

לאחר השגת הדאטה, מבצעים סימולציית הרצה עבור אלגוריתמי $scheduling$ שונים.



1.2.5 מימוש

הרעיון הבא הינו פשוט־ ליישם את אלגוריתם ה־ $scheduling$ ולבדוק אותו עבור מערכת אמיתית. אך מדובר בשיטה בעלת מחיר גבוה (בזמן ומשאבים) וסיכון גבוה, שכן כל בדיקה ושיפור דורשים הרצה מחדש על המערכת. כמו כן ייתכן שהסביבות בהן רוצים לבדוק את האלגוריתם שונות. אך $schdulers$ גמישים יותר יכולים להשתנות לפי אתר או לפי מערכת או לכלול $APIs$ לשינוי עדיפויות.

2 Chapter 6 - Synchronization Tools

2.1 Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

2.2 Objectives

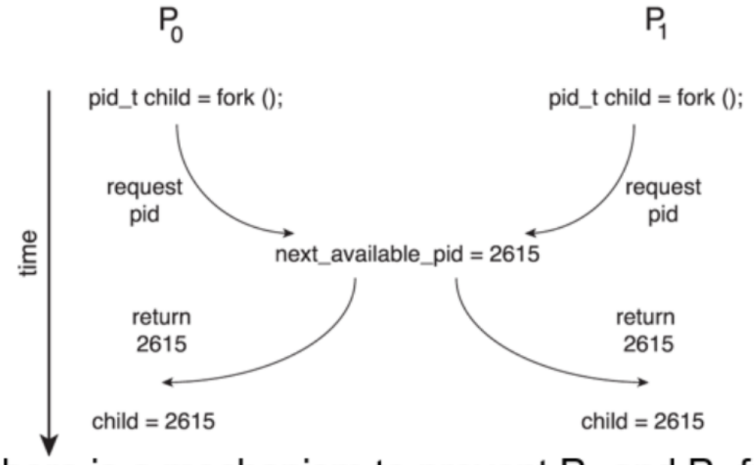
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

2.3 בעיית ה־ Critical Section

כפי שלמדנו, תהליכים יכולים לרוץ באופן מקביל, כמו כן תהליך יכול להיעצר במהלך ריצתו לפני שסיים. במקרים אלה, אם התהליכים חולקים דאטה, יש לוודא כי סדר הקריאה וכתיבה בה נשמר. כאשר שני תהליכים יכולים לרוץ, ולא ניתן לדעת מי מהם יבוצע קודם או ישתלט כעת על ה־ CPU, מתקבל מצב $race\ condition$. להלן דוגמה; נניח שישנם n תהליכים המסומנים $p_1, p_2, ..., p_n$. לכל תהליך סגמנט בקוד שהוא האזור הקריטי שלו. למשל, ייתכן שהתליכים משנים פרמטרים משותפים, מעדכנים טבלה, כותבים בטבלה וכו'. כאשר תהליך נמצא בקטע קריטי, אסור שאף אחד אחר יהיה בו. בעיית האזור הקריטי היא לעצב פרוטוקול אשר מאפשר זאת.
כל תהליך חייב לבקש רשות להיכנס לאזור קריטי ב־ $entry\ section$, לאפשר לתהליך אחר להיכנס לאזור קריטי ב־ $exit\ section$ ואז לעבור ל־ $remainder\ section$.
להלן רשימת דרישות מסודרת לפתרון לבעית האזורים הקריטיים;

1. $Mutual\ Exclusion$ - אם תהליך P_i רץ באזור הקריטי שלו, אף תהליך אחר לא יכול לרוץ בו.

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

2. *Progress* - אם אף תהליך לא מבצע רץ באזור הקריטי שלו וקיים תהליך שמבקש לעשות זאת, אז בחירת התהליך הבא שמורשה להיכנס לאזור הקריטי לא יכולה להידחות.
3. *Bounded Waiting* - חייב להיות חסם למספר הפעמים שתהליכים אחרים יכולים להיכנס לאזור הקריטי לאחר שתהליך ביקש לעשות זאת, עד אשר הוא אכן נכנס.

עד כאן הדרישות. נעבור לדבר על הפתרונות עצמם.

2.4 פתרון מבוסס *Interrupts*

מדובר ברעיון פשוט. כאשר תהליך ב־ *Entry section*, נשבית *interrupts* וכאשר הוא ב־ *Exit section* נחזיר אותם.

אז הבעיה הראשונה שניתן לשים אליה היא שאין אף מנגנון *Bounded waiting*. כמו כן, מה אם קיימים שני *CPUs*?

2.5 פתרון תוכנה 1

זהו פתרון המיועד לשני תהליכים. נניח ששפת המכונה של ה־ $load$ וה־ $store$ הן אטומיות כך שלא ניתן להפריע להן (עם $interrupts$) בשפה זו, שני התהליכים חולקים משתנה יחיד; $int\ turn$. אותו משתנה מציג לאיזה מהתהליכים מותר כעת לגשת לאזור הקריטי. מיצג פשוט לפתרון:

```
while (true){

        while (turn = = j);

        /* critical section */

        turn = j;

        /* remainder section */

}
```

כמובן שמתקיים $mutual\ exclusion$. עם זאת, אין הבטחה לקיום שתי הדרישות השניות...

2.6 $Peterson's\ Solution$

כמו מקודם, נניח ששפת המכונה של ה־ $load$ וה־ $store$ היא אטומית. אך כעת, לכל זוג תהליכים ישנם שני פרמטרים משותפים (!)

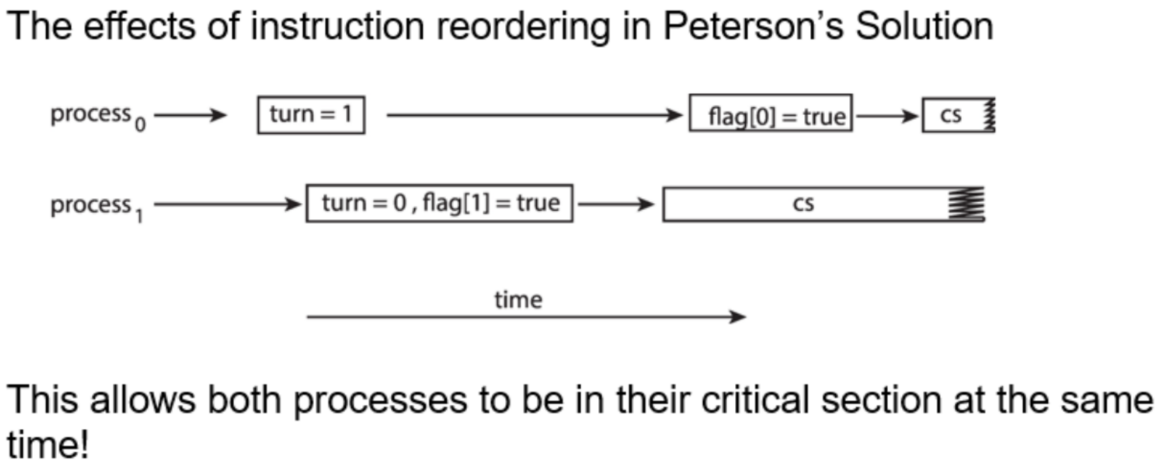
- $int\ turn$
- $boolean\ flag[2]$

הפרמטר $turn$ מציג את התהליך שכעת יכול להיכנס לאזור הקריטי. $flag$ מציג לכל תהליך האם הוא מבקש להיכנס לאזור (אם $flag[i] = true$) וכמובן הסבר ויזואלי של האלגוריתם; פתרון זה מקיים את כל שלושת

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

הדרישות :O

אבל - הוא לעתים לא פותר בעיות במערכות מודרניות. זאת מכיוון שבמערכות הפעלה מודרניות נהוג שכאשר ישנן שתי פקודות בלתי תלויות, ניתן לבצע אותן בכל סדר $(reordering)$ כלומר, אם ישנם מספר חוטים בכל תהליך, ייתכן שהם יבוצעו אחד אחרי השני. אך אז ייתכן שיקרה המצב הבא;



כדי להתאים את הפתרון של פטרסון, משתמשים בחסם זיכרון $(memory\ barrier)$
מודל זיכרון הוא הזיכרון אשר ארכיטקטורת מחשב מספקת לתוכנות האפליקציות. מודל זיכרון יכול להיות;

- $Strongly\ ordered$ - כאשר שינוי בזיכרון במעבד אחד מיידית נראה בשאר המעבדים.
- $Weakly\ ordered$ - כאשר שינוי בזיכרון של מעבד אחד יכול לא להיות נראה באופן מיידי אצל שאר המעבדים.

מחסום זיכרון הוא פקודה שמכריחה כל שינוי בזיכרון להיות נראה אצל כל שאר המעבדים (נקרא $propagate$)
כאשר פקודת מחסום זיכרון מבוצעת, המערכת מוודאת שכל ה- $loads$ וה- $stores$ הושלמו לפני שנדרשים השלמים חלקיים. לכן, גם אם התרחש $reorder$, מחסום הזיכרון מוודא שפעולות ה- $store$ הושלמו בזיכרון וניתן לראות אותם במעבדים האחרים לפני העלאות עתידיות או פעולות $store$ נוספות.
להלן דוגמה של חסם זיכרון בפעולה; שימו לב - עם חסם זיכרון הפכנו את פתרון פטרסון לחומרתי. נראה

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs
```
while (!flag)
  memory_barrier();
print x
```
- Thread 2 now performs
```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

דוגמאות נוספות לפתרונות חומרתיים;

2.7 חומרת סינכרון

מערכות רבות מספקות תמיכה חומרתית למימוש אזורים קריטיים.
למשל, *uniprocessors*, מעבדים בהם כל חוט רץ באופן סדרתי, היכולים להשבית *interrupts*. קוד הרץ כעת יכול להתבצע ללא *preemption*. אך לרוב שיטה זו איננה יעילה במערכות *multiprocessors*.

נסתכל על שני פתרונות אחרים־ פקודות מכונה ומשתנים אטומיים. אך כל זאת ועוד, בשבוע הבא :)