מערכות הפעלה - הרצאה 7

שרון מלטר, אתגר 17

10 ביולי 2024

1 Chapter 4 Cont.

1.1 Implicit Threading

ככל שיותר חוטים נדרשים, כך מסובך יותר לוודא נכונות תוכנה בניהולם כאשר משתמשים בכל אחד באופן מפורש (explicit) יצירה ואחזוק של החוטים נעשה על ידי המעבדים וה־ $run-time\ libraries$ ולא על ידי המתכנתים. ישנן מספר שיטות ל־ $implicit\ threading$

1.1.1 Thread Pools

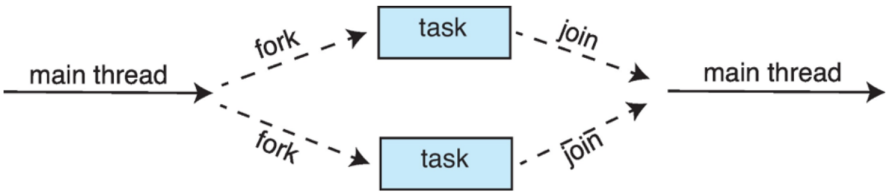
במודל זה יוצרים מספר חוטים בבריכה שם הם מחכים לעבודה. יתרונות השיטה:

- בדרך כלל מהיר יותר בהענקת שירות לאחר בקשה, אין צורך ליצור חוט חדש לכך.
- מספר החוטים באפליקציה (או אפליקציות) חסום במספר החוטים בבריכה.
- הפרדת משימה שיש לבצע מהמכניקה של יצירת חוט מאפשרת אסטרטגיות שונות לביצוע אותה המשימה. כלומר, ניתן לתכנן את תזמון המשימות מעת לעת.

ה־ API של ווינדוס תומך בבריכות חוטים:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

1.1.2 Fork − Join



1.1.3 OpenMP

$OpenMP$ הוא סט של הנחיות למעבד ו־ API ל־ $C, C++$. סט זה מאפשר תכנות מקבילים בסביבות של שיתוף־זיכרון. בפרט, הוא מזהה בלוקים של קוד שהם אזורים מקבילים ($parallel\ regions$) אשר יכולים לרוץ במקביל.

1.1.4 *Grand Central Dispatch*

לא רלוונטי :)

1.1.5 *Intel Threading Building Blocks*

לא רלוונטי :) - יותר חשוב שנבין שישנן מספר שיטות להתייחס ל- *implicit threading*

כעת נדבר על מספר סוגיות אשר יוצרים חוטים.

1.2 סוגיות עם חוטים

- סמנטיקה של $fork()$ ו- $exec()$
- *signal handling* - סינכרוני ואסינכרוני.
- ביטול חוט מסוים - אסינכרוני או דחוי.
- אחסון של חוט.
- *Schedular Activations*

נעבור על הסוגיות בהרחבה.

1.2.1 *Semantics of $fork()$ and $exec()$ System Calls*

האם $fork()$ צריכה להכפיל רק את החוט הקורא או את כולם? (כלומר האם להכפיל את כל החוטים הקיימים של התהליך ההורה) כדי לפתור את הבעיה, ב- *Unix* ישנן שתי גירסאות של $fork()$.
$exec()$ בדרך כלל עובד באופן נורמלי- מחליפים את התהליך הנוכחי וגם את כל החוטים שלו.

1.2.2 *Signal Handling*

סיגנלים משומשים ב- *Unix* משומשים כדי לעדכן תהליך שאירוע התרחש. *signal handler* משומש כדי לעבד סיגנלים.

1. הסיגנל נוצר על ידי אירוע מסוים.
2. הסיגנל נשלח לתהליך.
3. הסיגנל מטופל על ידי אחד משני *handlers*; ברירת מחדל או אחד שהוגדר על ידי המשתמש.

לכל סיגנל יש *default handler* שהקרנל מריץ כאשר מטפלים בסיגנל והיוזר יכול להעמיס אותו בעזרת *handler* שהוא מגדיר. כאשר מדובר במערכת *single − threaded*, הסיגנל מועבר לתהליך- אך מה לגבי *multi − threaded*?
ישנן מספר גישות לאן להעביר את הסיגנל כאשר ישנם מספר חוטים;

- להעביר את הסיגנל לחוט שהוא רלוונטי אליו.
- להעביר את הסיגנל לכל חוט שבתהליך.
- להעביר את הסיגנל לחוטים מסוימים בתהליך.
- להקצות חוט שמטרתו לקבל את כל הסיגנלים שנשלחים לתהליך.

1.2.3 *Cancellation of Target Thread*

זהו המקרה בו יש לבטל חוט לפני שהוא סיים את ריצתו. החוט שיש לבטל נקרא *target thread*. ישנן שתי גישות מרכזיות לטיפול בבעיה זו:

- ביטול אסינכרוני - מבטלים את חוט המטרה באופן מיידי.
- *Deferred cancellation* - חוט המטרה יכול לבדוק מעת לעת האם יש לבטל אותו.

להלן קוד $Pthread$ ליצירת וביטול חוט; שימי לב שכאשר מזמנים ביטול חוט רק מבקשת לבטל אותו,

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

הביטול עצמו תלוי במצב החוט־ אם הביטול של חוט לא מופעל, הביטול מחכה עד שהחוט מאפשר זאת. סוג

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

ברירת המחדל הוא $deferred$. הביטול יכול לקרות רק כאשר החוט מגיע לנקודת ביטול, $pthread_testcancel()$ $i.e,$ ואז מזומן $cleanup\ handler$. במערכות $Linux$, ביטול חוטים מטופל דרך סיגנלים.

1.2.4 $Thread-local\ Storage\ TLS$

TLS מאפשר לכל חוט להחזיק בהעתק שלו של הדאטה. יכולת זו שימושית כאשר אין לנו שליטה על תהליך יצירת החוט (כלומר כאשר משתמשים בבריכת חוטים) לא מדובר במשתנים לוקאליים, מכיוון שמשתנים לוקאליים קיימים רק בזמן ביצוע פונקציה יחידה וה־ TLS זמין לגישה מפונקציות שונות. עם זאת, TLS דומה לנתון סטאטי והוא ייחודי לכל חוט.

1.2.5 $Scheduler\ Activations$

לא רלוונטי מאוד. יש לדעת רק כי קיים חיבור בין חוטי היוזר לחוטי הקרנל ומכנים אותו $lightweight\ process$, ממרחב היוזר חיבור זה נראה כמו מעבד, מכיוון שדרכו הוא מקבל משאבי CPU.

Chapter 5 CPU Scheduling 2

Outline 2.1

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives 2.2

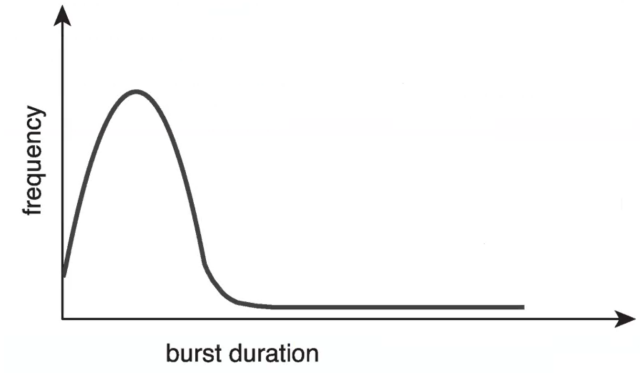
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms

נתחיל ממושגים שיש להבין.

2.3 מושגים בסיסיים

- ניצול מקסימלי של CPU מוקנה עם $multiprogramming$.
- $CPU - I/O\ Burst\ Cycle$ - ריצת תהליך כוללת מעגל CPU ואז $I/O\ wait$
- לאחר $CPU\ burst$ ישנו $I/O\ burst$
- כדאי מאוד לדעת כיצד מתלפגים ה- $CPU\ bursts$

להלן היסטוגרמה המתארת מה קורה בזמן *CPU burst*; הרכיב אשר בוחר אילו תהליך ישתמש ב־ *CPU*,



הינו, כפי שצוין בפרקים הקודמים, ה־ *CPU Scheduler*.

2.4 *CPU Scheduler*

ה־ *CPU scheduler* בוחר תהליך מבין התהליכים שנמצאים ב־ *ready queue* ומקצה ליבת *CPU* לאחד מהם.
ייתכן שיהיה צורך בקבלת החלטה של *CPU scheduler* כאשר תהליך;

1. מחליף ממצב *running* למצב *waiting*
2. מחליף ממצב *running* למצב *ready*
3. מחליף ממצב *waiting* למצב *ready*
4. מבוטל (*terminates*)

בסיטואציות 1 ו־4, אין החלטה במושגי *scheduling*. תהליך חדש (אם קיים אחד ב־ *ready queue*) חייב להיבחר לריצה. אך כן קיימת בחירה בסיטואציות 2 ו־3.

נציג את אפשרויות הבחירה שלנו.

2.5 *Preemptive and Nonpreemptive Scheduling*

בסיטוצאיות 1 ו־4, תכנית התזמון היא *nonpreemptive*. אחרת, היא *preemptive*.
ב־ *Nonpreemptive scheduling*, כאשר ה־ *CPU* הוקצה לתהליך, התהליך שומר אותו עד שהוא משחרר על ידי סיום פעולותיו או על ידי החלפה למצב *waiting*.
כמעט כל מערכות ההפעלה היום, בין היתר ווינדוס, *Linux*, *MacOS*, ו־ *Unix* משתמשות באלגוריתמים שהינם *preemptive scheduling*.
כמובן שב־ *preemptive* ייתכן שישנו מצב של *race condition*, כאשר נתונים משותפים בין מספר תהליכים; נניח שיש שני תהליכים שחולקים אותה דאטה. תהליך 1 רץ וקורא מאותה דאטה, אבל אז מתקיים *preemption* כך שתהליך 2 מתחיל לרוץ והוא משנה את אותה הדאטה. כאשר תהליך 1 ימשיך בקריאה, לא ניתן לדעת מה השתבש (כמו בדוגמה בהרצאה קודמת) נחקור את הנושא הזה בפירוט בפרק 6.
בחזרה לנושא שלנו, *Dispatcher module* נותן לתהליך שנבחר על ידי ה־ *CPU scheduler* את השליטה על ה־ *CPU*. פעולה זו מערבת;

- *Switch context*
- החלפה ל־ *user mode*
- קפיצה למקום הנכון בתכנית היוזר כדי להתחיל אותה מחדש.

המושג *Dispatch latency* מתאר את הזמן שלוקח ל־ *dispatcher* לעצור תהליך אחד ולהתחיל תהליך אחר.
כעת נלמד כיצד אנו 'שופטים' את ה־ *scheduling*.

2.5.1 *Scheduling Criteria*

להלן מספר תכונות אשר לפיהן ניתן לשפוט טיב ה־ *scheduling*:

- *CPU utilization* ־ השארת ה־ *CPU* עסוק כמה שניתן.
- *Throughput* ־ מספר התהליכים שמשלימים את ריצתם ביחידת זמן.
- *Turnaround time* ־ כמות הזמן שלוקח לבצע תהליך מסוים. כלומר, הזמן שעובר מהרגע בו תהליך הגיע ל־ *ready queue* עד שהוא סוים.
- *Waiting time* ־ כמות הזמן שבו תהליך חיכה ב־ *ready queue*
- *Response time* ־ כמות הזמן שלוקח מהרגע שבו בקשה נשלחה עד שנוצרה תגובה ראשונה.

אלגוריתם *scheduling* הינו אופטימלי אם הוא מקיים;

- *Max CPU utilization*
- *Max throughput*
- *Min turnaround time*
- *Min waiting time*
- *Min response time*

כעת נלמד כמה אלגוריתמי *scheduling* ונשווה ביניהם.

2.5.2 *First – Come, First – Served (FCFS) Scheduling*

השיטה הכי פשוטה, בה התהליך הראשון המגיע ל־ *ready queue* מקבל ראשון את ה־ *CPU* ומשחרר אותו כאשר הוא עובר ל־ *waiting state* או מסיים את ריצתו. להלן דוגמה לאלגוריתם בפעולה; במקרה ה־ *waiting time*

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

הממוצע הוא 17.

נראה דוגמה שנייה; כאן ה־ $waiting\ time$ הממוצע הוא 3 (!)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case

הסיבה שבדוגמה השנייה קיבלנו יעילות הרבה יותר גבוהה, היא ה־ $Convoy\ effect$; תהליך קצר שנמצא בטור אחרי תהליך ארוך.

2.5.3 $Shortest-Job-First\ (SJF)\ Scheduling$

נחבר לכל תהליך את האורך של ה־ $CPU\ burst$ הבא שלו. בעזרת אורכים אלה נבחר את התהליך בעל אורך הזמן הקצר ביותר. ב־ SJF ישנו את ה־ $average\ waiting\ time$ המינימלי. איך יודעים לחבר את הזמנים? ־ שואלים את היוזר על אורך התהליכים או משערכים. ניתן לעשות זאת באמצעות אורכי ה־ $CPU\ bursts$ הקודמים של התהליך וממוצע אקספוננציאלי:

1. t_n = actual length of nth CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

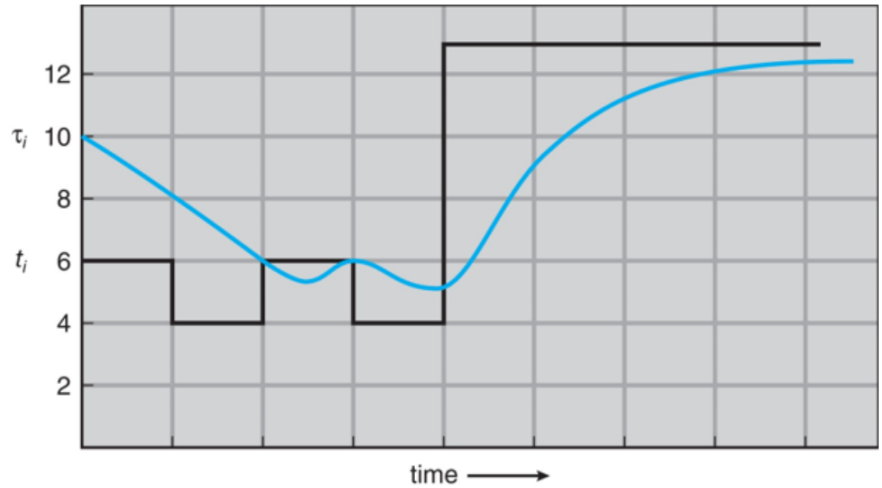
כאשר לרוב בוחרים $\alpha = \frac{1}{2}$. להלן דוגמה:

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$$

- Since both α and $(1-\alpha)$ are less than or equal to 1, each successor predecessor term has less weight than its predecessor

וכאן גרף המייצג את השערות ה־ $CPU\ bursts$ לעומת הערך האמיתי שלהם;

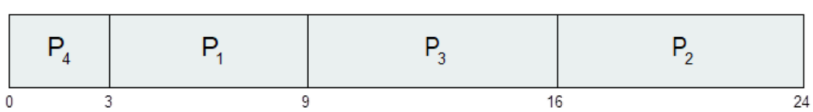


CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

בגירסא בה ישנו $preemption$ האלגוריתים נקרא $shortest-remaining-time-first$, מכיוון שכאשר מגיע תהליך שזמנו קצר יותר ל־ $waiting\ queue$ מתרחש $preemption$ והוא מתחיל לרוץ.
נראה דוגמה;

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

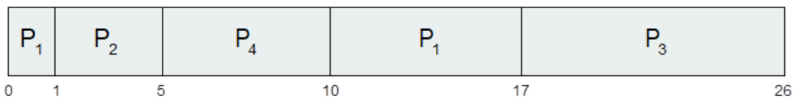
2.5.4 $Shortest\ Remaining\ Time\ First\ Scheduling$

כפי שצוין מקודם, זוהי הגירסא עם $preemption$ של SJN.
דוגמה לאלגוריתם בפעולה;

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



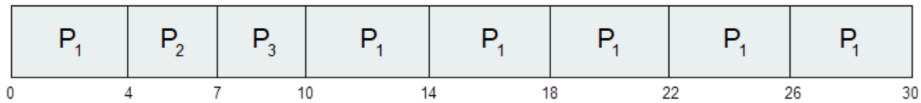
- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

2.5.5 $Round\ Robin\ (RR)$

CPU, המכונה $time\ quantum$ ונסמנה q. לרוב היא מסדר גודל של $10-100$ מילישניות. אחרי שהזמן חלף, התהליך $preempted$ ועובר לסוף ה־ $ready\ queue$. ה־ $Timer$ מודיע מתי עבר הזמן. q צריך להיות גדול ביחס ל־ $context\ switch$ אחרת ה־ $overhead$ של ההחלפה גדול מדי על מנת להיות יעיל.
אם ישנם n תהליכים ב־ $ready\ queue$, אז כל תהליך מקבל $1/n$ מזמן ה־ CPU במקבצים של לכל היותר q יחידות זמן בפעם אחת. אף תהליך לא מחכה יותר מ־ $(n-1)q$ יחידות זמן.
שימי לב שככל ש־ q גדל האלגוריתם יותר דומה ל־ $FIFO\ (FCFS)$.
נראה דוגמה של RR בו $q=4$; כמו כן, להלן גרף של ה־ $turnaround\ time$ (הזמן שלקח מהרגע בו תהליך

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

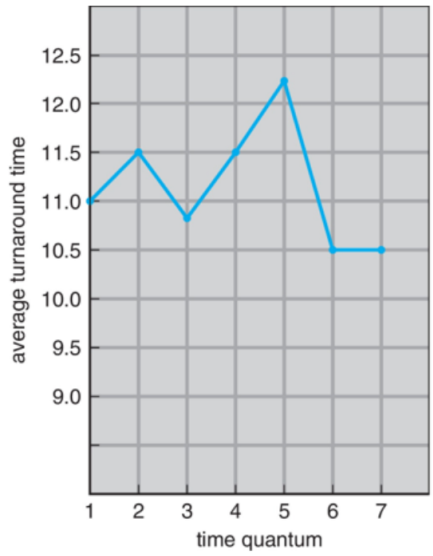


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

הגיע ל־ $ready\ queue$ עד שהוא סוים) כתלות ב־ $quantum\ time$; $CPU\ burst$ הינו הזמן שבו תהליך משתמש ב־ CPU באופן רציף.

2.5.6 $Priority\ Scheduling$

לכל תהליך ניתנת עדיפות, שהינה מספר שלם. ה־ CPU מוקצה לתהליך בעל העדיפות הגבוהה ביותר, כאשר מספר עדיפות קטן יותר מייצג עדיפות גבוהה יותר. גישה זו יכולה להיות $preemptive$ או $nonpreemptive$.
ניתן לומר שהאלגוריתם של SJF הינו זהה לאלגוריתם זה בו העדיפות ניתנת על סמך אורך ה־ $CPU\ burst$ הבא



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

של התהליך.
בעיה שניתן לשים אליה לב באלגוריתם היא שתהליך בעל עדיפות נמוכה עלול להגיע להרעבה, $starvation$, בו הוא עלול לעולם לא לרוץ. הפתרון לכך הוא הזדקנות, $aging$, כלומר מתן עדיפות לתהליך הנמצא ב־ $queue$ $ready$ ככל שעובר זמן.
דוגמה לפעילות האלגוריתם;

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

- Average waiting time = 8.2

עכשיו ננסה לשלב בין $priority\ scheduling$ ל־ RR;

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

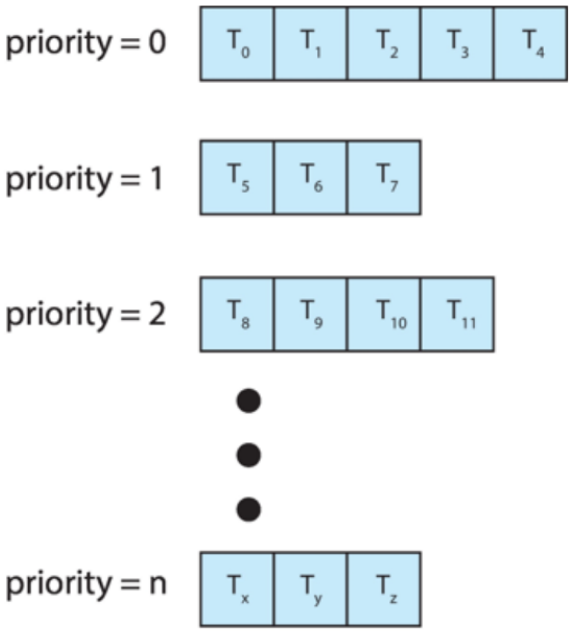
- Gantt Chart with time quantum = 2

2.5.7 *Multilevel Queue*

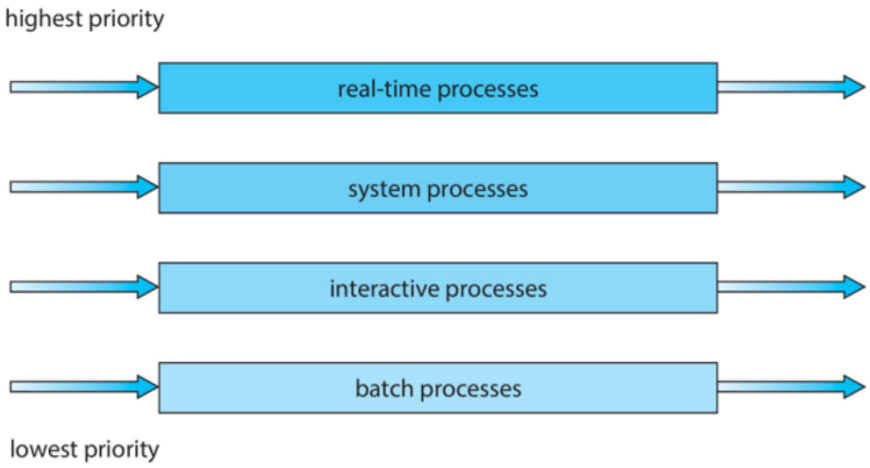
בפתרון זה ה־ *ready queue* מורכב מכמה תורים שונים. באופן כללי, *Multilevel queue scheuler* מאופיין בתכונות הבאות:

- מספר התורים
- אלגוריתם ה־ *scheduling* ששייך לכל תור
- שיטה אשר מחליטה לאיזה תור ייכנס תהליך שנכנס למצב *ready*
- *scheduling* בין התורים (מאיזה תור לוקחים תהליך שיוקצה עבורו ה־ *CPU*)

למשל, ניתן להעניק עדיפות לכל תור כך שתהליך בעל עדיפות זהה יוכנס אליו. בכל פעם ייבחר תהליך מהתור בעל העדיפות הכי גבוהה.

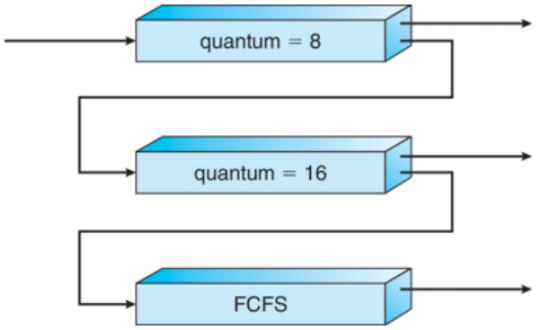


ניתן למשל לבחור את עדיפותו של כל תהליך לפי סוגו:



כאשר תהליך $batch$ הוא תהליך המשמש את המחשב לחישובים שיש לבצע באופן רפיטיטיבי פעמים רבות ברקע.
דוגמה לביצוע האלגוריתם; עד כה דיברנו על $process\ scheduling$, אך מה לגבי $threads\ sheduling$?

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



2.6 $Thread\ Scheduling$

כאשר ישנה תמיכה במספר חוטים לתהליך ($multithreaded$), ה־ $scheduling$ מתייחס רק לחוטים ולא לתהליכים. כאשר כמובן יש הפרדה בין חוטים השייכים למרחב היוזר וחוטים ממרחב הקרנל (אחרת המערכת חסרת הגנה) במודלים של $Many-to-One$ ו־ $Many-to-Many$, ספריית החוטים מתזמנת את חוטי רמת היוזר כך שירוצו על LWP ($Light\ Weight\ Process$). זהו מעבד וירטואלי שמבצע $scheduling$ עבור חוטי יוזר. כל חוט־יוזר מוצמד לחוט־קרנל, ועבורם מערכת ההפעלה מבצעת $scheduling$ ל־ $CPUs$ האמיתיים. ה־ LWP נקרא גם $process-contention\ scop\ (PCS)$ מכיוון שהתחרות על הקצאת $CPUs$ היא בתוך התהליך (בין החוטים). לרוב התחרות נקבעת לפי עדיפות שקבע המתכנת.

תחום ה־ $scheduled$ של חוטי הקרנל מכונה $system-contention\ scope\ (SCS)$ מכיוון שהתחרות על המשאבים נעשית בין כל חוטי המערכת (הרי חוטי היוזר מוצמדים לחוטי קרנל)

נעבור לדבר על $scheduling$ בסוג הספריות החביב עלינו, $Pthread$.

2.7 $Pthread\ Scheduling$

ה־ API מאפשר לסווג בין SCS ל־ PCS בזמן יצירת חוט (על מנת לוודא היכן צריך התחום להתחרות ועל פי איזה $scheduling\ algorithm$) מבצעים זאת בעזרת הערכים $PTHREAD\_SCOPE\_SYSTEM$, $PTHREAD\_SCOPE\_PROCESS$ עם זאת לא כל מערכת הפעלה מכבדת זאת. ב־ $Linux$ ו־ $macOS$ יש אפשרות לבחור ב־ $PTHREAD_SCOPE_SYSTEM$ בלבד.
להלן ה־ API של $Pthread\ Scheduling$;

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

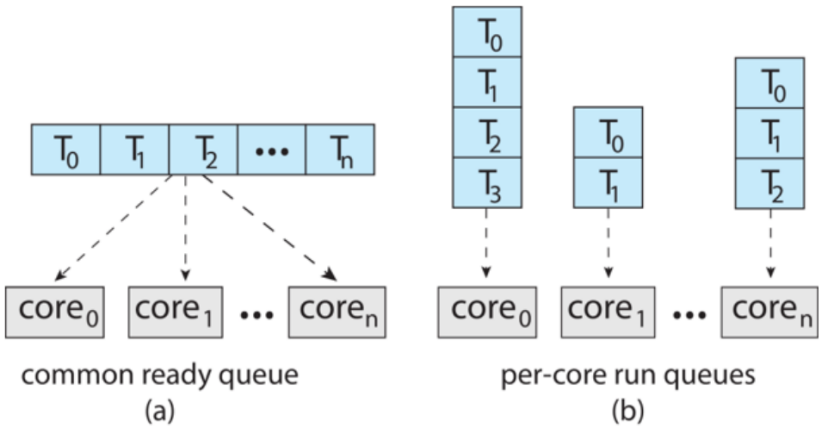
עד כאן תזמון חוטים.
נעבור על ארכיטקטורות שונות עבורן ניתן להשתמש במספר מעבדים.

2.8 *Multiprocessor Scheduling*

כמובן שיותר מסבך לבצע *CPU scheduling* כאשר ישנם מספר מעבדים שניתן להשתמש בהם. נלמד כמה ארכיטקטורות אשר מתמודדות עם בעיה זו.

2.8.1 *Multicore CPUs*

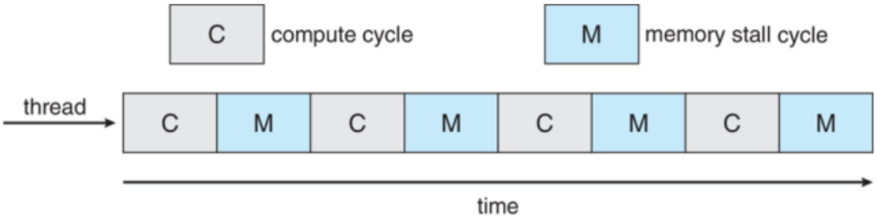
Symmetric multiprocessing (*SMP*) הינו כאשר כל מעבד מבצע ניהול זמנים בעצמו. כל החוטים יכולים להיות באותו ה־ *ready queue* או שלכל מעבד יש את ה־ *ready queue* שלו.



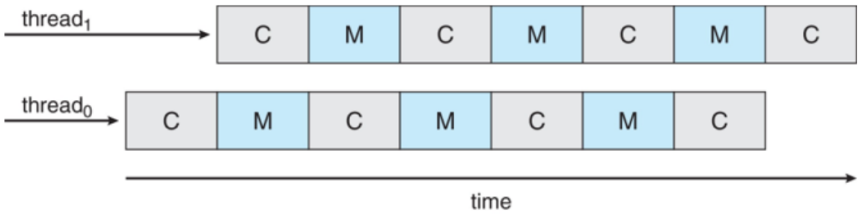
רעיון נוסף הצובר תאוצה הוא להציב מספר ליבות מעבדים באותו השבב הפיזי, כך שהחישוב בהם מהיר יותר ודורש פחות כוח.

2.8.2 *Multithreaded cores*

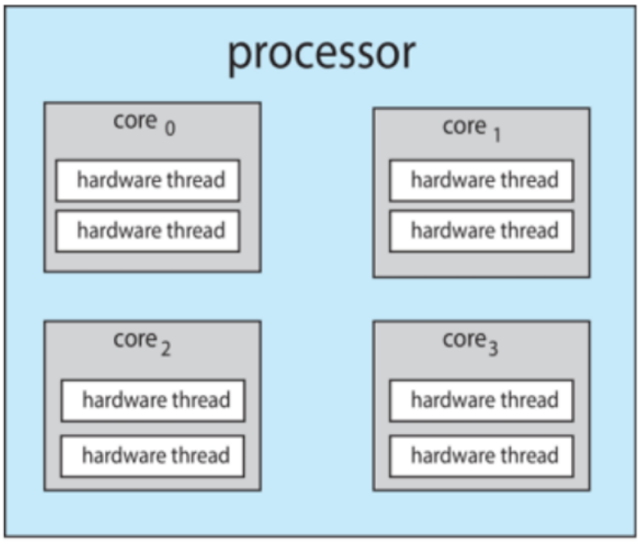
כמו כן הרעיון של מספר חוטים לכל ליבה גם גדל־ רעיון זה מנצל *memory stall* (זמן הנאבד כאשר ניגשים לזיכרון) על מנת להתקדם בחוט אחר בזמן ששליפת הזיכרון נעשית. להלן תרשים המייצג זאת;

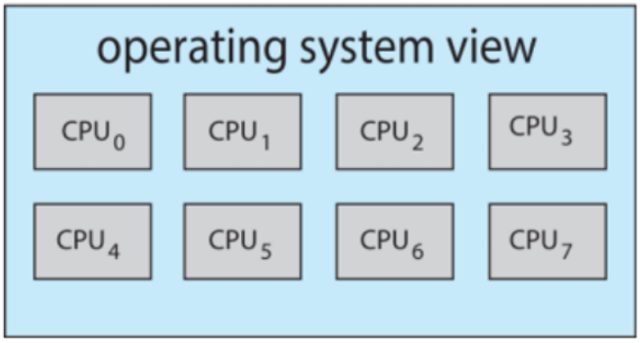


כאן רואים מערכת התומכת ב־ *multithread* (עבור חוטי חומרה) בשרטוט זה מעבר ל־ *multithread*, ישנם מספר החוטים אשר שייכים לכל ליבת מעבד, כך שבכל פעם אשר מתרחש *memory stall* מחליפים לחוט אחר על מנת לאפשר התקדמות; ניתן לקרוא לגישה זו גם $Chip-multithreading\ (CMT)$, וב־ *Intel* היא מכונה



hyperthreading. להלן דוגמה של מערכת התומכת בגישה;





במקרה זה ישנן ארבע ליבות למעבד. אם לכל אחת מהן ניתן להצמיד שני חוטי חומרה, אז מערכת ההפעלה רואה בהן כ־ 8 מעבדים לוגיים (כלומר ניתן להריץ 8 חוטים) זאת מכיוון שזהו מספר החוטים אשר ניתן להקצות להם ליבת מעבד.
בסך הכל נקבל שישנן שתי רמות של *scheduling*, אחת אשר משדכת את חוטי התכנה למעבדים לוגיים, ושנייה אשר משדכת חוטי חומרה לליבות הפיזיות.
נכיר מושגים חדש הרלוונטי לגישה זו. כאשר תומכים ב־ *SMP*, יש לוודא שכל המעבדים יוצרים לוח זמנים יעיל וגם החוטים מתחלקים ביניהם באופן אופטימלי. המושג *Load balancing* מתאר ניסיון לחלק באופן שווה את עומס העבודה.
Push migration זוהי משימה שיש לבצע מעת לעת המוצאת את העומס הניתן לכל מעבד. אם אין איזון, בין המעבדים הוא דוחף משימות ממעבד עמוס למעבדים אחרים.
Pull migration ־ תהליכי *idle* מוציאים תהליכים ב־ *waiting state* ממעבד עסוק.

מושג נוסף: *Processor Affinity*.
כאשר חוט רץ על מעבד אחד, תוכן ה־ *cache* של אותו המעבד נגיש על ידי אותו החוט. מכנים חוט זה כאחד שיש לו *affinity* (זיקה) לאותו המעבד, "*processor affinity*". איזון העמסה (*load balancing*) יכול להשפיע על זיקה למעבד כאשר חוט מועבר מאותו מעבד, אך אותו החוט מעבד את התוכן שהיה לו באותו ה־ *cache*.
Soft Affinity ־ מערכת ההפעלה מנסה לשמור על חוט שיישאר באותו המעבד, אף לא מבטיחה זו. *Affinity Hard* ־ מערכת ההפעלה מאפשרת לכל תהליך להגדיר סט של מעבדים שהוא יוכל לרוץ עליהם.

2.8.3 *NUMA systems*

אם מערכת ההפעלה פועלת בגישת *NUMA*, היא תשדך חוטים לזיכרון הכי קרוב למעבד עליו הם רצים.

כעת נעבור לדבר על דרישה מיוחדת וההשלכות שלה על *scheduling*.

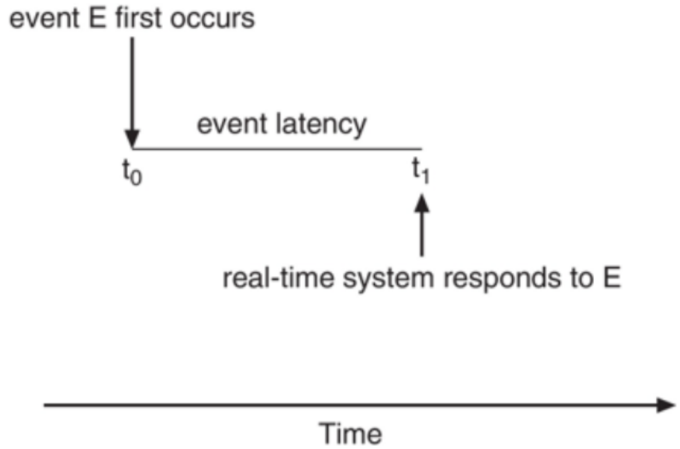
2.9 *Real – Time CPU Scheduling*

- *Soft real – time systems* ־ משימות בעלי קריטיות גבוהה יהיו בעלות העדיפות הגבוהה ביותר, אך אין הבטחה למתי הן יבוצעו.
- *Hard real – time systems* ־ המשימה חייבת להיות מבוצעת עד הדדליין שלה.

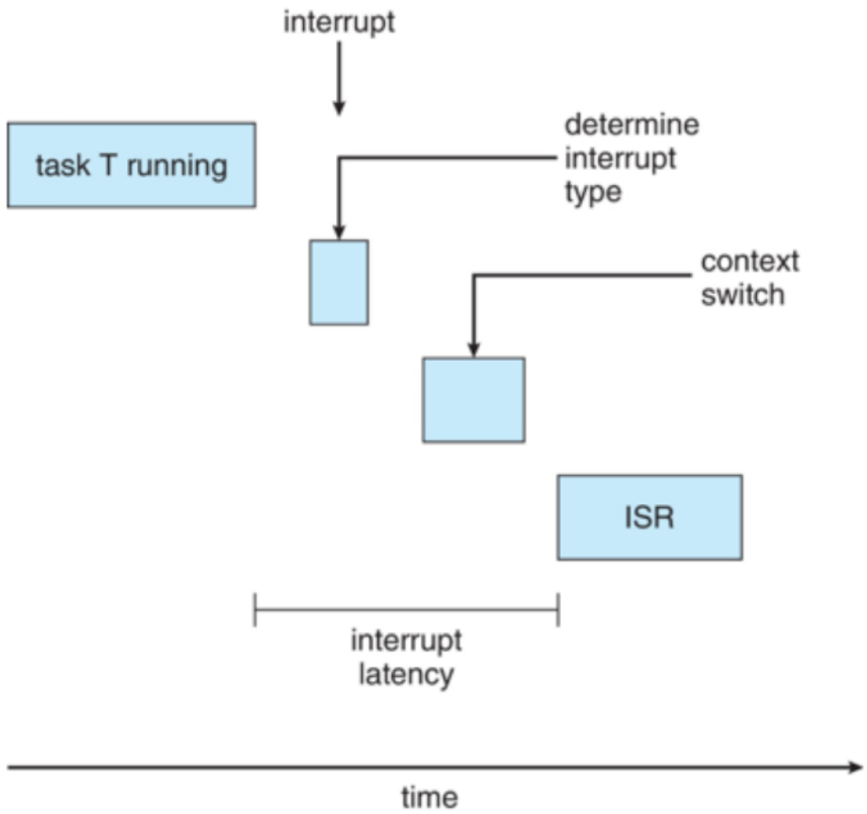
כדי להתמודד עם הדרישה הרלוונטית, נלמד 3 מושגים חדשים;

- *Event latency* ־ כמות הזמן שחלף מהרגע שבו אירוע קורה עד שהוא מטופל.
- *Interrupt latency* ־ הזמן שעובר מהגעת *interrupt* עד להתחלת הטיפול בו.
- *Dispatch latency* ־ הזמן שלוקח ל־ *scheduler* לבצע *context switch*

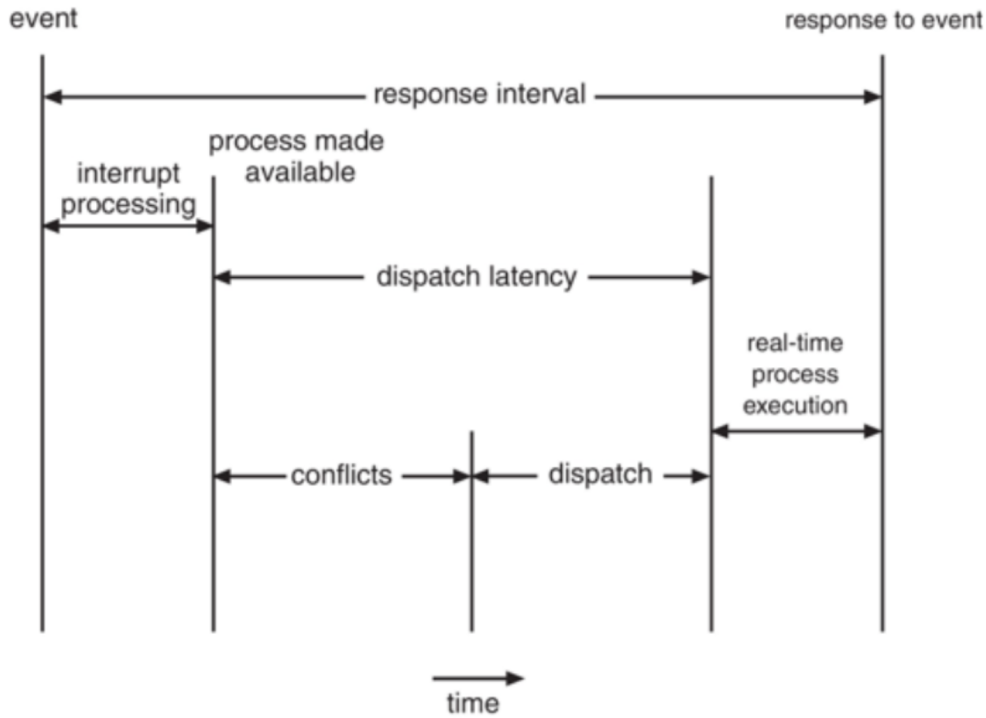
להלן תרשים פשוט המציג מהו *event latency*;



וכמו כן תרשים עבור *Interrupt latency*;



ו־ *Dispatch latency*; חלק הקונפליקט של ה־ *dispatch* מורכב משני חלקים:



1. *preemption* של התהליך הנוכחי שרץ ב־ *kerne mode*

2. שחרור משאבים של תהליך בעדיפות נמוכה ונזקקים על ידי תהליך עם עדיפות גבוהה.

כפי שצוין, עבור $real-time\ scheduling$, אנו נעזרים בגישה של עדיפויות. עם זאת, ישנה חובה גם לעמוד בדדליינים של התהליכים הקריטיים, (עבור $hard\ real-time$) לכן עצם התמיכה בעדיפות איננה מספקת. לכל תהליך נעניק תכונות חדשות: תהליך יוכל להיות $periodic$.
תהליכים אלה יכולים לדרוש הקצאת CPU למשך אינטרוולים קבועים. ניתן לעמוד בדדליין כאשר ישנו זמן עיבוד t, דדליין d ותקופה p, ומתקיים $0 \leqslant t \leqslant d \leqslant p$. כמו כן ה־ $rate$ של ה־ $periodic\ task$ הוא $1/p$, כלומר ישנה לכל היותר משימה קריטית בכל תקופה. להלן תרשים המדגים את ההגדרות;