**שאלה 1 (40 נקודות)**

בשאלה זו נדון באלגוריתם הזימון של לינוקס במערכת מרובת מעבדים ובפרט באלגוריתם המשמש את לינוקס לאיזון העומס בין המעבדים, לצורך פתרון השאלה עליכם להיעזר בספר Understanding The Linux Kernel 3rd Edition, פרק 7.

1. האם יש הבדל בין ביצוע השגרה schedule במערכת עם מעבד יחיד לבין ביצועה במערכת מרובת מעבדים? אם יש הבדל הסבר מהו ואם לא הסבר מדוע.  
   **תשובה:**

ההבדל בין שגרת schedule במערכת עם מעבד יחיד לבין מערכת מרובת מעבדים הוא שבמערכת מרובת מעבדים schedule קוראת לפונקציה load\_balance() אשר תפקידה לאזן את מספר התהליכים בין runqueues של המעבדים הנוספים במערכת, דבר אשר אינו רלוונטי במערכת בעלת מעבד יחיד.

1. כפי שראינו בתרגולים, לכל מעבד ישנו runqueue משלו והתהליך הבא שיבחר לרוץ על המעבד נבחר מתוך אותו runqueue. פרט יתרון אחד וחסרון אחד של שיטה זו.  
   **תשובה:**

יתרון של השיטה – לתהליך יש זיכרון שהוא צריך לקרוא ממנו / לכתוב אליו. כפי שלמדנו, כאשר תהליך ניגש לזיכרון המעבד "מקרב" את הזיכרון אליו ע"י שימוש ב-cache. בהפרדת ה-rnuqueues לפי מעבדים, אנו מגדילים את הסיכוי שלתהליך הבא שייבחר לרוץ יש את איזורי הזכרון שלו במעבד שאותו הוא יקבל.

בנוסף, אם לכל מעבד יש runqueue משלו, הוא יכול לשמור אותו ב-cache הרלוונטי ובכך להקטין משמעותית את משך זמן ביצוע אלגוריתמי תזמון למיניהם (אלגוריתמים שדורשים גישה לרשימת התהליכים, כגון schedule). אם ה-runqueue היה משותף לכל המעבדים, בכל בדיקת החלפת הקשר היה צורך לגשת לזיכרון משותף (איטי) בכל החלפת הקשר, וזוהי תקורה יקרה.

חסרון של השיטה – כאשר יש מספר רב של תהליכים מוכנים לריצה ב runqueue מסויים כאשר ה runqueues האחרים ריקים מתהליכים , במקרה זה אנחנו מנצלים את היכולת של מעבד יחיד, מה שגורם לחוסר ניצול – כאשר תהליכים רבים מחכים בתור לקבל זמן מעבד. באופן כללי אם כל התהליכים במערכת היו ברשימה אחת, מערכת ההפעלה הייתה יכולה לקבל החלטה "נכונה יותר" לגבי איזה תהליך להריץ על המעבד הנוכחי (ועל כל המעבדים).

1. הסבר מהו scheduling domain וכיצד הוא משמש באלגוריתם איזון המשימות.  
   **תשובה:**

scheduling domain הם בעצם קבוצות של מעבדים (פיזיים או לוגיים) – שמחולקות לפי רמות – כאשר ברמה הגבוהה ביותר – נמצאת קבוצת כל המעבדים, בכל רמה מתחת קיימת תת קבוצה של מעבדים בהתאם לרמה.

באופן זה ניתן לבצע את איזון העומס של תהליכים בין המעבדים מבוצע באופן יעיל יותר – האיזון מתבצע בין מעבדים באותו domain.

1. הסבר כיצד מחליט האלגוריתם האם יש צורך בהפעלת איזון משימות על המעבד.  
   **תשובה:**

בכל פסיקת שעון, האלגוריתם מטפס במעלה היררכית ה-scheduling domains ובודק האם הגיע הזמן לבדיקה ותיקון איזון מבין ה-groups ב-domain הנוכחי. הפרמטרים הקובעים האם הגיע הזמן תלויים הם 1. האם המעבד הנוכחי פנוי (מריץ את idle), 2. האם יש חוסר איזון בין ה-groups ברמה כזו שדורשת קריאה לפונקציה מאזנת (צריך שזה יעבור threshold מסוים, אחרת יהיו קריאות רבות לפונקציות יקרות בזמן וכך נתעסק יותר באיזון בין מעבדים לבין עבודת מעבדים) ו-3. פרמטרים נוספים תלויי scheduling domain. המחשבה כאן היא למנוע קריאות תכופות מידי ל-load\_balance, אבל כן לדעת באיזה scheduling domains כדאי לאזן יותר ובאילו פחות.

1. הסבר כיצד מאזן האלגוריתם את המשימות בין המעבדים, האם ישנו איזון מלא או הקלה בעומס בלבד?  
   **תשובה:**

האלגוריתם מטפס במעלה ההיררכיה של ה-scheduling domain, ובכל שלב בודק האם יש חוסר איזון מבין ה-groups בתוך ה-domain הנוכחי. אם אין חוסר איזון ב-domain הנוכחי, מטפסים למעלה ל-domain הבא עד שמגיעים לקבוצת כל המעבדים במערכת. ע"מ לאזן עומס (במקרה שיש חוסר איזון בין groups), נקראת פונקציה load\_balance.

פונקציה זו מחפשת את הקבוצה (group) העסוקה ביותר בתחום (scheduling domain) – אם קבוצה זו שונה מהקבוצה אליה שייך המעבד הנוכחי ויש צורך באיזון בין הקבוצות בתחום, הפונקציה מחפשת את המעבד העסוק ביותר בקבוצה העסוקה ביותר ומנסה להעביר מספר תהליכים למעבד הנוכחי.

אם מעבר זה נכשל, הפונקציה מעירה תהליך במעבד העסוק שמנסה למצוא מעבד פנוי (תוך טיפוס במעלה ההיררכיה, החל מה-scheduling domain של המעבד העסוק) ולהעביר למעבד הפנוי תהליכים מהמעבד העסוק.

**שאלה 2 (20 נקודות)**

קבע האם הטענה נכונה או לא ונמק, תשובות ללא נימוק לא יתקבלו.

1. בעקבות שימוש מרובה בפקודה wait תהליך יכול להפוך לאינטראקטיבי.  
   **תשובה:**

הטענה נכונה – על מנת להפוך לתהליך אינטראקטיבי ,יש לעבור את הבדיקה הבאה

-bonus <= -DELTA - לכן עלינו לקבל בונוס שהוא שלילי ככל האפשר- על מנת לקבל בונוס כזה –

יש למקסם את ה sleep\_avg ככל האפשר – זה מתבצע בעזרת חישוב של דלתא הזמן שהתהליך לא היה activate . בפקודה wait אנחנו הופכים את התהליך ל TASK\_INTERRUPTIBLE – לכן בהגעה לפונקציה schedule שם אנחנו מבצעים switch case על state כאשר יבחר deactivate\_task – מה שיגרום לכך שעל מנת להחזיר את התהליך ל runqueue נצטרך להשתמש ב activate\_task() , ששם אנחנו מעדכנים את ה sleep avg של תהליך.

לכן, אם נבצע wait ונחכה לאירוע שייקח לו זמן להתרחש (יצרנו בן עם משימה ארוכה, למשל) אז בהחלפת הקשר כתוצאה מקריאה ל-schedule נצא מה-runqueue באמצעות deactivate\_task. בתוך schedule, הערך של sleep\_timestamp יעודכן לזמן הנוכחי. לאחר זמן ארוך (הבן יסיים) נחזור, וכשנחזור תיקרא פונקציית activate\_task והערך של sleep\_avg יעודכן לערך יחסית גדול, מה שיגדיל כאמור את הבונוס.

1. בעקבות שימוש מרובה בפקודה sched\_yield תהליך יכול להפוך לאינטראקטיבי.  
   **תשובה:**

הטענה לא נכונה – כפי שהוסבר לעיל , תהליך צריך לצאת מהמעבד( להפוך ללא ניתן לריצה) על מנת שנוכל להחזיר אותו עם activate\_task ולעלות לו את ה sleep avg – ב sched\_yield – אנחנו לא משנים את ה state של תהליך לכן בהגעתו ל schedule הוא עדיין מוגדר מוכן לריצה ולא יבצע deactivate\_task.

**שאלה 3 (20 נקודות)**

1. הסבר מדוע לא ניתן להשתמש בקריאה ישירה ל - schedule בפונקציה scheduler\_tick ויש צורך להדליק את הדגל need\_resched כאשר תהליך מסיים את ה - time\_slice שלו.  
   **תשובה:**

כאשר אנו ב-scheduler\_tick זה אומר שאנו בתוך פסיקת שעון – זוהי פסיקת חומרה שלא reentrant. כלומר, אם לא נצא ממנה באופן מסודר לפני שניכנס אליה שוב, המערכת לא תהיה יציבה (עלולים להיות באגים). אם באמצע scheduler\_tick נקפוץ ל-schedule ונחליף הקשר, אנחנו עלולים להיכנס שוב ל-scheduler\_tick עם התהליך החדש לפני שיצאנו ממנו עם התהליך הישן. לכן צריך לסיים את הטיפול בפסיקה (בפרט, לסיים את scheduler\_tick ולחזור לקוד המקורי) ולפני החזרה לקוד של המשתמש ואחרי סיום הטיפול נעבור ל-schedule (שזה בעצם מטרת הדגל need\_resched)

1. הסבר מדוע במאקרו switch\_to קופצים לפונקציה \_\_switch\_to בעזרת הפקודה jmp ולא בעזרת הפקודה call  
   **תשובה:**

בתחילת המאקרו ביצענו שמירה של רגיסטרים רלוונטיים (לתהליך המחולף). לאחר ביצוע הפונ' \_\_switch\_to אנחנו נצטרך לחזור ולשחזר את אלה של התהליך החדש (כפי שנשמרו בפעם הקודמת שהוא הוחלף). אבל זה רלוונטי רק להחלפות הקשר להתהליכים שכבר רצו. אם ההחלפה התבצעה לתהליך שנוצר עכשיו, הוא צריך להמשיך לרוץ לאחר החלפת ההקשר ממקום אחר לחלוטין (ret\_from\_fork).לכן, הקפיצה לפונקציה תתבצע בעזרת jmp – כלומר ללא ערך חזרה. אם מדובר בתהליך חדש, נמשיך כרגיל, אבל אם מדובר במקרה של שני תהליכים "ותיקים" נחזור ללייבל ששמרנו מראש על ראש המחסנית (הלייבל 1), שמסמל כתובת לשחזור אותם רגיסטרים.

**שאלה 4 (20 נקודות)**

אלגוריתם lottery scheduling הינו אלגוריתם זימון הסתברותי. כל תהליך במערכת מקבל מספר כלשהו של "כרטיסי הגרלה" (לכל כרטיס מספר יחודי); בכל שלב, האלגוריתם מגריל מספר של אחד מכרטיסי ההגרלה שחולקו, והתהליך שמחזיק בכרטיס הגרלה זה מקבל את המעבד לפרק זמן קצוב (תמיד אותו פרק זמן).

1. נניח כי לאלגוריתם יש 20 כרטיסים. כיצד עליו לחלק אותם בין ארבעה תהליכים (A, B, C, D) כך שהתהליכים יקבלו 10%(A), 5%(B), 60%(C), 25%(D) בהתאמה?  
   **תשובה:**

A – 2 כרטיסים, B – כרטיס אחד, C – 12 כרטיסים, D – 5 כרטיסים

1. כיצד ניתן להגיע באמצעות אלגוריתם זה לביצועים הדומים לאלו המושגים על-ידי אלגוריתם round-robin?

**תשובה:**

באלגוריתם RR התהליכים מסתדרים במעגל – כאשר תהליך אחד רץ ומסיים את הקוואנטום שלו הוא

מושהה והתהליך הבא בתור מורץ. כאשר מגיעים לתהליך שממנו התחלנו מסיימים epoch.

על כן על מנת להגיע לביצועים דומים יש לחלק לכל תהליך מספר שווה של כרטיסי הגרלה, באופן הזה

כל תהליך יקבל זמן מעבד שווה – לכן בסה"כ ניתן לדמות את האופן ריצתם של התלהיכים באלגוריתם זה לאלגוריתם RR.

1. איך ניתן להגיע באמצעות אלגוריתם זה לביצועים הדומים לאלו המושגים על-ידי אלגוריתם

Shortest remaining time to completion first?   
(ניתן להניח כי יש preemption, התהליכים לא מגיעים יחד, אבל זמן הביצוע ידוע כאשר התהליך מגיע.)  
**תשובה:**

לפי אלגוריתם Shortest remaining time to completion first , תהליך שנותר לו זמן הכי קצר לביצוע המשימה שלו , יקבל את המעבד ראשון. לפי הנתון זמן הביצוע של כל תהליך ניתן כאשר התהליך מגיע

על כן יש לבצע חישוב של ניתנת מספר כרטיסי הגרלה באופן דינמי לפי זמן הביצוע.

לכן (נסמן את החסם העליון על זמן ביצוע תהליך שנמדד ב-ticks כ-L) לכל תהליך שמגיע שמדווח שהוא רץ x זמן, ניתן לו L-x+1 כרטיסים. בכל פסיקת שעון: ניתן לתהליך הפוסק כרטיס אחד נוסף, עד למקסימום של L. התוצאה של זה היא שלכל תהליך יש L-x+1 כרטיסים (כאשר x הזמן שנותר לו לרוץ) גם בכל רגע ולא רק בתחילת הריצה, אלא אם כן תהליך רץ יותר משדיווח (ואנו מניחים שזה לא המצב).

כך, תהליך קצר מאוד (נניח, זמן ריצה 1) יקבל L כרטיסים ולכן בסיכוי גבוה מאוד יקבל את המעבד; לעומתו תהליך שרוצה לרוץ L זמן יקבל כרטיס 1 ולכן הסיכוי שהוא ייבחר נמוך מאוד.

1. איזה יתרון יש לשיטה הזו על פני shortest remaining time to completion first אמיתי?  
   **תשובה:**

בגרסה ההסתברותית, אין מצב שבו תהליך **בטוח** ירעב, כי לכל תהליך – אפילו לתהליך הארוך ביותר – יש לפחות כרטיס אחד. לכן, גם אם יש עוד 100 תהליכים שלכל תהליך יש L כרטיסים, ובכל יחידת זמן מגיע תהליך חדש ומקבל L כרטיסים, תמיד קיים סיכוי שהתהליך עם כרטיס אחד (התהליך הארוך) יקבל את המעבד. לעומת זאת, בשיטה הלא-הסתברותית נוכל ליצור מצב שיירה ולהרעיב את התהליך הארוך.