מערכות הפעלה

סיכום החומר בקורס "מערכות הפעלה" בטכניון

סיכם: אור גלעד

המסמך נכתב על ידי **אור גלעד**

מערכות הפעלה – סיכום החומר למבחן

מבוא

מערכת הפעלה – שכבת תוכנה המנהלת ומסתירה פרטים של חומרת המחשב, מספקת לאפליקציה אבסטרקציה של מכונה וירטואלית ייעודית וחזקה ומנהלת את משאבי המערכת ומשתפת אותם בין תהליכים, תוכניות ומשתמשים.
מערכת ההפעלה מאפשרת להריץ אפליקציות תוך כדי הבטחת נכונות (ע"י יצירת גבולות לזיכרון, עדיפויות בין תהליכים וייצוב המערכת כולה) ומספקת נוחיות שימוש (ע"י הסתרת פרטים, תיאום בין התהליכים וההתקנים ושימוש במערכת קבצים).
כל אפליקציה רוצה את כל המשאבים הקיימים: זמן מעבד, זיכרון, קבצים, אמצעי O\I ושעון המערכת.

מבנה המחשב – ההתנהגות של מערכת ההפעלה מוכתבת ע"י החומרה שעליה היא רצה, כאשר החומרה יכולה לפשט וגם לסבך משימות של מערכת ההפעלה.
מערכת ההפעלה נעזרת במספר מנגנוני חומרה שתומכים בה: שעון חומרה, פעולות סנכרון אטומיות, פסיקות, קריאות מערכת ההפעלה, פעולות בקרה על קלט/פלט, הגנת הזיכרון ופעולות מוגנות.
פקודות מוגנות – חלק מפקודות המכונה מותרות רק למערכת ההעפלה כמו גישה לרכיבי קלט/פלט, שינוי של מבני הנתונים לגישה לזיכרון, עדכון של סיביות מצב מיוחדות לקביעת עדיפות טיפול בפסיקות או פקודת halt. הארכיטקטורה תומכת בשני מצבים לפחות (kernel mode, user mode) כאשר תוכניות משתמש רצות ב-user mode ומערכת ההפעלה מריצה את הפקודות שלה ב- kernel mode ורק במצב הזה המעבד מבצע פקודות מוגנות.
קריאת מערכת (system call) – הדרך שבה משתמש קורא לפרוצדורה של מערכת ההפעלה. תהליך משתמש מבצע קריאת מערכת על מנת לבקש מהגרעין לבצע עבורו שירות כלשהו. השירות חייב להתבצע ע"י הגרעין אם הוא מצריך גישה לחומרה או למבני נתונים של מערכת ההפעלה או של תהליכים האחרים לצורך יצירת תהליכים חדשים, תקשורת בין תהליכים או גישה לנתוני משתמשים.
נוצרת קריאה לפסיקה (מספר ה-system call המבוקש מוצב ברגיסטר eax, מכין פרמטרים לפי השירות המבוקש וביצוע פקודת int 0x80), קריאת המערכת מזהה ע"י פרמטר, מצב התוכנית הקוראת נשמר, מבצעים וידוא הפרמטרים שהועברו, מבצעים את רוטינת השירות בתוך הגרעין וחוזרים לתוכנית הקוראת בסיום.
הגנה על הזיכרון – מערכת ההפעלה צריכה להגן על תוכניות המשתמשים, זו מפני זו, ולהגן על עצמה מפניהם. השיטה הפשוטה היא להשתמש ברגיסטרים (base ו-limit) לכל תוכנית, שמגדירים חלוקה קבועה של גבולות הריצה של גישות התוכניות לזיכרון. במערכות מודרניות משתמשים בזיכרון וירטואלי שמדמה זיכרון גדול יותר, כך שתוכניות יכולות לגשת לכתובת זיכרון שאינן קיימות בפועל.

ארגון מערכת ההפעלה – הגרעין שבשימוש היום הוא גרעין מודולרי שמשתמשים בו במודולים דינמיים שניתן לטעון ולהסיר אותם מהזיכרון לפי דרישה. בצורה הזאת מונעים מהגרעין להתנפח ללא צורך בזיכרון המחשב ומתאפשר לטעון רכיבי קוד ונתונים לפי דרישה וכך להשיג ביצועים טובים יותר.

ארכיטקטורת IA32

ארכיטקטורה של 32 ביט שנמצאת במעבדי אינטל החל מ-1985. הארכיטקטורה מגדירה למערכת 3 מצבי עבודה, כאשר העיקרי מביניהם הוא Protected Mode שכולל תמיכה מלאה ב-32 ביט, ניהול זיכרון, ריבוי משימות וכו' (שאר המצבים מהווים תאימות לארכיטקטורות ישנות יותר).

הארכיטקטורה מבדילה בין 4 רמות שונות של פעולות מכונה מיוחסות:

- Ring 0 – הרמה המקסימלית שבה ניתן לגשת לכל החומרה במחשב ללא הגבלה, כאשר רוב מערכות ההפעלה משתמשות ברמה הזו כ-kernel mode.
- Ring 3 – הרמה המינימלית שמאפשרת גישה מוגבלת, בעיקר לחלקי זיכרון בלבד. רמה זו היא רמת קוד המשתמש (user mode).

כאשר המעבד מבצע קוד, ה-ring שבו נמצא המעבד נקרא Current Privilege Level (CPL).

ניהול זיכרון – הארכיטקטורה מסוגלת לנצל עד 4GB של זיכרון פיזי. לכל יישום (כולל מע' ההפעלה) יש זיכרון "פרטי" משלו שרק הוא יכול לגשת אליו, על מנת להגן על נתוני מערכת ההפעלה ועל הנתונים של כל יישום ויישום, כאשר למע' ההפעלה מותר לגשת לזיכרון של היישום.
הזיכרון מאורגן בצורת "איזורים", כאשר כל יישום יכול לגשת לשני איזורים לפחות: אחד לקוד ואחד לנתונים ולמחסנית. כאשר קוד מבקש לגשת לאיזור, החומרה מאשרת את הגישה במגבלות הבאות: לפי הפעולה המבוקשת (קריאה, כתיבה, הרצה), כאשר לכל איזור מוגדרות הרשאות המגבילות את סוג הפעולה, ולפי ה-CPL של המבקש, על מנת למנוע מיישומי המשתמש לגשת לאיזורים של מערכת ההפעלה. הדברים נעשה בעזרת הגדרת Descriptor Privilege Level (DPL) לכל איזור, וניתן לגשת לאיזור רק אם ה-DPL שלו גדול או שווה ל-CPL של המבקש.
כשניגשים לזיכרון, הקוד מספק כתובת לוגית המכילה segment:offset. ה-segment הוא מספר בן 16 ביט המזהה את איזור הזיכרון המבוקש, ואילו ה-offset הוא מספר בן 32 ביטים המזהה כתובת ספציפית בתוך האיזור המבוקש. החומרה מתרגמת את הכתובת הלוגית לכתובת פיזית (מספר בן 32 ביט) של תא בזיכרון המחשב.

רגיסטרים – מרבית הרגיסטרים על מעבד IA32 הינם בני 32 ביט ונתן לגשת למילה (16 ביט) בהשמטת ה-e בתחילת השם.

- 4 רגיסטרים לחישוב כללי: eax, ebx, ecx, edx. ניתן לגשת גם לבית הנמוך (8 ביט) בכל רגיסטר אם מחליפים את ה-x ב-l, ולבית השני הנמוך ביותר ע"י החלפת ה-x ב-h.
- 2 רגיסטרים כלליים המשמשים גם לגישה למערכים: esi, edi.
- מצביע המחסנית esp.
- רגיסטר כללי המשמש גם בקריאה לפונקציות ebp.
- מצביע התוכנית eip.
- Segment registers – רגיסטרים בני 16 ביט המכילים את הסגמנט בגישה לזיכרון: cs(code), ds(data), ss(stack), es/fs/gs.
- רגיסטר הדגלים eflags.

הדגלים ב-eflags מתחלקים לשני סוגים: **חיווי סטטוס** (מתעדכנים בקביעות כל פעולת מכונה ומשמשים לחיווי מצב המעבד) ו**מתגי בקרה** (משנים מצב ע"פ פעולה מפורשת של הקוד על מנת להשפיע על התנהגות המעבד בהמשך).
דגלי חיווי סטטוס הינם **CF** (דולק אם בחישוב האחרון נוצא נשא carry מעבר לסיבית הגבוהה ביותר), **PF** (דולק אם מספר הסיביות במצב 1 בתוצאת החישוב האחרונה הינו זוגי), **ZF** (דולק אם תוצאת החישוב האחרונה היא 0).
מתגי הבקרה הינם **IF** (Interrupt Flag – כאשר דולק הוא מאפשר למעבד לקבל ולטפל בפסיקות חומרה חיצוניות), **IOPL** (I\O Privilege Flag – קובע את ה-ring המקסימלי עבורו מותר לבצע פעולות קלט/פלט, כך שרק קוד שרץ ב-CPL=0 יכול לשנות את הערך).

מחסנית – לכל יישום במערכת יש מחסנית המשמשת לקריאה לפונקציות במהלך ביצוע הקוד.
המחסנית נגישה דרך צמד הרגיסטרים ss:esp המצביע על האיבר הנוכחי שבראש המחסנית.
המחסנית גדלה בכיוון הכתובות הנמוכות, כאשר בכל פעולה push, esp מוקטן ב-2 או ב-4 גם אם

מכניסים למחסנית פחות מ-16 ביטים, כתובת segment:offset נשמרות במחסנית ע"י דחיפת ה-segment כערך של 32 ביט ואחריו ה-offset.

קריאה לפונקציה ב-Linux

1. תחילה מכניסים את הפרמטרים של הפונקציה למחסנית בסדר הפוך (הפרמטר הימני ביותר ראשון).
2. כעת מתבצעת הוראת המכונה call שדוחפת את כתובת החזרה (ערכו של elp) למחסנית וקופצת לתחילת הפונקציה ע"י הצבת כתובת תחילת הפונקציה ב-eip. אם קוראים לפונקציה באותו איזור זיכרון, מכניסים למחסנית את ה-offset בלבד ומעדכנים רק את ערכו של elp (ולא cs).
3. הקוד של הפונקציה הנקראת חייב לשמור על ערכי הרגיסטרים ebp, esi, edi, ebx במקרה שהוא משתמש בהם, ולשחזר את ערכם בסיום ביצוע הפקודה.
4. הפונקציה הנקראת שומרת תחילה את ebp במחסנית ואז מציבה את ebp להצביע על ראש המחסנית (שמכיל את ערכו הקודם של ebp). לאחר מכן נשמרים שאר הרגיסטרים לפי הצורך.
5. אחרי שמירת הרגיסטרים מוקצה על המחסנית מקום למשתנים המקומיים של הפונקציה הנקראת ע"י הקטנת ערכו של esp בגודל המקום הנדרש. כך מובטח שבחזרה מהפונקציה ישוחררו אוטומטית כל המשתנים המקומיים.

לרגיסטר ebp יש תפקיד מיוחד בקריאה לפונקציה: הוא מצביע על בסיס מסגרת הפונקציה, כלומר על המקום במחסנית בו מתחילה שמירת הנתונים של הפונקציה הנקראת. בהתאם לכך, כל מיקומי הפרמטרים והמשתנים המקומיים של הפונקציה נגישים כערכים יחסיים לערכו של ebp (למשל: הערך של ebp-16 מצביע על המשתנה המקומי הראשון של הפונקציה).
לפני סיום ביצוע הפונקציה משוחזרים ערכי הרגיסטרים שנשמרו ע"י הפונקציה הנקראת. בסיום ביצוע הפונקציה מתבצעת פקודת ret ששולפת את כתובת החזרה לקוד הקורא מהמחסנית וקופצת לכתובת זו ע"י הצבתה ל-eip. הקוד הקורא אחראי לשמור את כל הרגיסטרים שבשימושו ושאינם באחריות הפונקציה הנקראת, והם משוחזרים לאחר החזרה מהפונקציה. הקוד גם אחראי לפינוי הפרמטרים ע"י הגדלת esp.
הפונקציה מחזירה ערך בסיום (אם יש כזה) לפי הכללים הבאים: ערך בגודל עד 8 ביט יוחזר ב-al, עד 16 ביט ב-ax, עד 32 ביט ב-eax (נפוץ) ועד 64 ביט הערך יוחזר לצמד edx:eax. אם הפונקציה אמורה להחזיר ערך שהוא רשומה מורכבת, המקום עבור גודל הערך יוקצה מראש על המחסנית ע"י הקוד הקורא לפונקציה כפרמטר חבוי, והפונקציה תעדכן את תוכן הרשומה במחסנית לפני סיום הביצוע.
המחסנית גדלה לכיוון הכתובת הנמוכות.

קריאות מערכת ב-Linux

קריאת מערכת ממומשת ע"י מעטפת קוד שמפעילה באמצעות פסיקת תוכנה את קריאת המערכת שמבצעת את השירות המבוקש ומחזירה את תוצאת ביצוע השירות לקוד שקרא לה, וכן מקושרת לתוכנית שמשתמשת בה. הערך המוחזר ע"י פונקצית המעטפת במקרה של הצלחת ביצוע השירות תלוי בסוג השירות המבוקש, ובמקרה של כישלון בדרך כלל מוחזר הערך 1- וסוג השגיאה יוחזר במשתנה הגלובלי errno.
כל קריאות המערכת בלינוקס מטופלות דרך פסיקת תוכנה שמספרה 128, כאשר סוג השירות המבוקש נקבע באמצעות מספר שירות המועבר ברגיסטר eax ע"י פונקצית המעטפת. בתגובה לפסיקה המעבד עובר מ-user mode ל-kernel mode ומפעיל בגרעין את שגרת הטיפול בפסיקה הקרויה system_call(). השגרה מנתבת את ביצוע הבקשה לפונקציה המתאימה לפי מספר השירות. system_call() פועלת תחת כללים שונים מכללי קריאה לפונקציה רגילה. אנו קופצים לשגרת טיפול

בפסיקה באיזור זיכרון של הגרעין, כאשר פעולת הקפויצה שומרת את eflags, cs, eip במחסנית של מערכת ההפעלה. החזרה מהשגרה היא בהוראה מיוחדת (iret) המשחזרת את הרגיסטרים האלו, מסגרת הפונקציה איננה מקושרת למסגרות קודמות, ואילו העברת הפרמטרים לשגרה היא באמצעות רגיסטרים בלבד. יש צורך בפרוטוקול קריאה מיוחד עבור system_call() כי במעברים בין user mode ו-kernel mode מתבצעת החלפת מחסניות.
ל-system_call() מעבירים את הפרמטרים הבאים: מספר השירות המבוקש באמצעות הרגיסטר eax, פרמטרים עבור השירות לפי הצורך ברגיסטרים הבאים (משמאל לימין): ebx, ecx, edx, esi, edi, ebp. פרמטרים הגדולים מ-32 ביט יועברו באמצעות מצביע לפרמטר ברגיסטר יחיד. אם נדרשים יותר מ-6 פרמטרים לשירות מעבירים רגיסטר יחיד המצביע לרשומה בזיכרון של התהליך המכילה את כל הפרמטרים.
sys_call_table – טבלה שכל כניסה בה מציין מספר השירות שמועבר ל-system_call(). מספר הכניסות לרוב מוגדר כ-256, אולם לא כל הכניסות מצביעות על פונקציות שירות פעילות, ואילו שאינן פעילות מצביעות על פונקצית הביצוע sys_ni_syscall(), המחזירה ENOSYS–.
פונקצית השירות כתובה ב-C, כאשר בודקים בה את התקינות של כל פרמטר שהועבר לה, בין אם הוא ייחודי לה, או אם המצביעים שניתנו לה אכן מצביעים על מיקום שמוכל בזיכרון של תהליך המשתמש.

שלבי הפעולה של system_call()

1. שמירת כל הרגיסטרים במחסנית – מתבצע על מנת להכין את הפרמטרים בתוך המחסנית כפי שפונקצית השירות מצפה להם.
2. טעינת מזהה תהליך המשתמש ל-ebx לצורך גישה לנתוני התהליך מתוך הגרעין.
3. בדיקת תקינות מספר השירות, על מנת לבדוק שהוא לא חורג מהטבלה.
4. הפעלת פונקצית ביצוע השירות לפי טבלת הכניסות. הערך שמוחזר מהפונקציה נשמר ברגיסטר eax.
5. סיום השגרה, ע"י הפעלת קטע קוד שמשחזר את כל הרגיסטרים מהמחסנית.

תהליכים

הגדרה – תהליך הוא יחידת הביצוע לארגון פעילות המחשב, יחידת הזימון ביצוע במעבד ע"י מערכת ההפעלה והוא גם תוכנית בביצוע. תהליך מאופיין ע"י מרחב כתובות, קוד התוכנית שרץ, הנתונים שבהם הוא משתמש, המחסנית שלו, השימוש שלו ברגיסטרים השונים והמספר שמייצג אותו. תהליך מריץ תוכניות, ותוכנית היא חלק ממצב התהליך, כאשר תוכנית יכולה לייצר כמה תהליכים. מבחינת מערכת ההפעלה, תהליך הינו ישות עצמאית הצורכת משאבים.

מרחב הכתובות – לכל תהליך יש אשליה שיש לו את כל מרחב הזיכרון בגדול 4GB, גם אם אין במחשב כמות זיכרון כזו, או שהיא תפוסה ע"י תהליכים אחרים. מרחב הכתובות כולל את הקוד של התהליך (עליו מצביע Program Counter), איזור למשתנים הגלובליים, heap שבה שומרים מקום לאובייקטים דינמיים ומחסנית התהליך (עליה מצביע Stack Pointer), כאשר שני האחרונים גדלים באופן דינמי.

כל תהליך נמצא באחד מהמצבים הבאים: מוכן לריצה, רץ (שניהם בטווח הקצר) או ממתין לריצה (טווח ארוך).

לכל תהליך יש Process control block (PCB) ששומר את כל הנתונים של התהליך. הוא נשמר כאשר התהליך מפונה ונטען כאשר התהליך מתחיל לרוץ. כאשר המעבד מפסיק להריץ את התהליך (כלומר מעביר אותו למצב של המתנה), ערכי הרגיסטרים נשמרים ב-PCB, וכאשר המעבד מחזיר את התהליך למצב ריצה, ערכי הרגיסטרים נטענים שוב במסגרת תהליך שנקרא **Context Switch**.

תורי מצבים – מערכת ההפעלה מחזיקה תורים של תהליכים, כאשר יש תור של תהליכים שמוכנים לריצה (ready) ותור של תהליכים שמחכים לרוץ (waiting). כאשר תהליך נוצר, מוקצה עבורו PCB והוא משורשר לתור המתאים, כאשר ה-PCB נמצא בתור המתאים למצבו של התהליך, והוא מועבר מתור לתור בהתאם למצב .בסיום התהליך, ה-PCB משוחרר.

יצירת תהליך – תהליך אחד יכול ליצור תהליך אחר, שיהיה הבן שלו. בדרך כלל, האב מגדיר או מוריש משאבים ותכונות לבניו, והוא יכול להמתין לבנו, לסיים או להמשיך לרוץ במקביל.
fork() – קריאת מערכת שיוצרת תהליכים ב-Unix. הפקודה יוצרת ומאתחלת PCB, מייצרת מרחב כתובות חדש ומאתחלת אותו עם העתק מלא של מרחב הכתובות של האב, מאתחלת משאבי גרעין לפי משאבי האב ומכניסה את ה-PCB לתור המוכנים. בשלב הזה יש שני תהליכים שנמצאים באותה נקודה בביצוע אותו קטע קוד. הבן מחזיר מ-()fork את הערך 0, ואילו האב מחזיר את מספר התהליך (pid) של הבן.
execv() – משמשת להפעלת תוכנית חדשה. הפקודה עוצרת את ביצוע התוכנית הנוכחית, מאתחלת את מצב המעבד וארגומנטים עבור התוכנית החדשה ומפנה את המעבד (ה-PCB מועבר לתור המוכנים). לא נוצר כאן תהליך חדש, ומשתמשים בפקודה כדי שתהליך הבן יריץ אפליקציה ששונה מתהליך האב.

תהליכים ב-Linux

באתחול Linux, גרעין מערכת ההפעלה יוצר שני תהליכים:

- Swapper (PID=0) – משמש לניהול הזיכרון, אולם ברוב המקרים המימוש שלו כולל רק את השורה while (1) כדי להשאיר את המחשב פועל במידה שאין תהליך אחר שרץ עליו.
- Init (PID=1) – ממנו נוצרים כל שאר התהליכים במערכת. התהליך ממשיך להתקיים לאורך כל פעולתה של מערכת ההפעלה.

כל תהליך נוסף נוצר כעותק של תהליך קיים, והתהליך החדש יכול לאחר היווצרו לבצע משימה שונה מאביו, באמצעות הסתעפות בקוד לאחר ההתפצלות מהאב, או ע"י טעינת תוכנית חדשה לביצוע (ע"י execv()).
תהליך אב יכול לבדוק סיום של כל תהליך בן שלו (אבל לא מעבר לזה). אב יכול להמתין לסיום בן לפני המשך פעולתו באמצעות קריאת המערכת ()wait. כדי לאפשר לאב לקבל מידע על סיום הבן, לאחר שתהליך מסיים את פעולתו הוא עובר למצב zombie שבו התהליך קיים כרשומת נתונים בלבד ללא שום ביצוע משימה, והרשומה נמחקת לאחר שהאב קיבל את המידע על סיום הבן. תהליך שאביו כבר סיים יהפוך להיות הבן של Init .

API לתהליכים ב-Linux

- Fork() – מעתיקה את תהליך האב לתהליך הבן וחוזרת בשני התהליכים. שני התהליכים ימשיכו לרוץ מאותה נקודה בקוד, כאשר גם הזיכרון והסביבה שלהם זהה. במקרה של כישלון, האב יחזיר 1-, ואם הפעולה הצליחה, האב יחזיר את ה-PID של הבן, והבן יחזיר 0. תהליך הבן הוא תהליך נפרד מתהליך האב לכל דבר, ולמרות שיש לשניהם את אותם משתנים בזיכרון, הם שמורים אצל שניהם בנפרד.
- Execv() – טוענת תוכנית חדשה לביצוע ע"י התהליך הקורא. מקבלת מסלול אל הקובץ המכיל את התוכנית ואת הארגומנטים שלו, ותחזיר 1- אם היה כשלון. אם הקריאה הצליחה, היא לא תחזור, היות ואיזורי הזיכרון של התהליך מאותחלים לתוכנית החדשה שמתחילה להתבצע מההתחלה.
- Exit() – מסיימת את ביצוע התהליך הקורא ומשחררת את כל המשאבים שברשותו. התהליך עובר למצב zombie עד שתהליך האב יבקש לבדוק את סיומו ואז הוא יפונה לחלוטין. היא

מקבלת את ערך הסיום שיוחזר לאב כאשר הוא יבדוק אם הפעולה הסתיימה, והיא אינה מחזירה נתונים.

- Wait() – גורמת לתהליך הקורא להמתין עד אשר אחד מתהליכי הבן שלו יסיים. אם אין בנים, או שכל הבנים כבר סיימו, הקריאה חוזרת מיד עם הערך 1-. אם יש בן שסיים ועד לא בוצע לו wait(), חוזרים מיד עם ה-PID שלו וסטטוס הסיום שלו, אחרת מחכים עד שבן כלשהו יסיים. ניתן להשתמש ב-waitpid() שממתין עד לסיום של בן מסוים.
- Getpid() – מחזירה לתהליך הקורא את ה-PID של עצמו. ע"י שימוש ב-getppid() ניתן להחזיר את ה-PID של תהליך האב של התהליך הקורא.

ניהול תהליכים בגרעין – לכל תהליך בלינוקס קיים בגעין מתאר תהליך (process descriptor) שהוא רשומה מסוג task_struct המכילה את המידע על התהליך: מצב, עדיפות, מזהה (PID), מצביע לטבלת איזורי הזיכרון של התהליך, מצביע לטבלת הקבצים הפתוחים של התהליך, מצביעים למתארי תהליכים נוספים שקשורים אליו, מצביעים למתאר של תהליך האב ושל קרובי משפחה נוספים, המסוף (מסך ומקלדת) שאליו התהליך מתקשר ועוד.
מצב התהליך נשמר בשדה state שהוא משתנה בגודל 32 ביט, כאשר בכל שלב בדיוק אחד הביטים דלוק בהתאם למצב התהליך. לכל תהליך יש את המצבים הבאים:

- TASK_RUNNING – התהליך רץ או מוכן לריצה.
- TASK_INTERRUPTIBE – התהליך ממתין לאירוע כלשהו אך ניתן להפסיק את המתנת התהליך ולהחזירו למצב ריצה.
- TASK_UNINTERUPTIBLE – התהליך ממתין לאירוע כלשהו, אבל לא ניתן להעיר אותו כל עוד האירוע הזה לא התרחש.
- TASK_STOPPED – ריצת התהליך נעצרה בצורה מבוקרת ע"י תהליך אחר (בדר"כ debugger).
- TASK_ZOMBIE – ריצת התהליך הסתיימה, אך תהליך האב של התהליך שסיים עדיין לא ביקש מידע על סיום התהליך באמצעות הקריאה wait(). התהליך קיים כמתאר בלבד.

מחסנית הגרעין – לכל תהליך יש מחסנית נוספת המשמשת את גרעין מערכת ההפעלה בטיפול באירועים במהלך ריצת התהליך, כמו פסיקות או קריאות מערכת. מחסנית הגרעין של כל תהליך מאוחסנת באיזור הזיכרון של הגרעין, ואם מתבצע מעבר בין user mode ו-kernel mode, מתבצעת החלפת מחסניות בין המחסנית הרגילה של התהליך למחסנית הגרעין שלו. ערכי ss:esp שמצביעים למחסנית הרגילה נשמרים ע"י המעבד במחסנית הגרעין מיד עם המעבר ל-kernel mode ומשוחזרים כשחוזרים ל-user mode.

ניהול קשרי משפחה בגרעין – קשרי המשפחה בין תהליכים מיוצגים בגרעין באמצעות מצביעים בין מתארי תהליכים. מתאר תהליך האב מצביע למתאר תהליך הבן הצעיר ביותר שלו (שנוצר אחרון) באמצעות השדה p_cptr במתאר התהליך. כל מתאר תהליך מצביע למתאר תהליך האב שלו באמצעות השדה p_opptr, או באמצעות השדה p_pptr שמצביע למתאר תהליך האב בפועל (הערכים שונים כאשר התהליך נמצא בריצה מבוקרת ע"י debugger). מתאר תהליך מצביע לאחיו הבוגר (האח שנוצר לפניו) באמצעות p_osptr, ואל אחיו הצעיר באמצעות p_ysptr.

רשימות מקושרות בגרעין – לצורך ניהול תורים ומבני נתונים אחרים גרעין משתמש ברשימות מקושרות כפולות מעגליות. האיברים ברשימה הם שדות המוכלים ברשומות מבני נתונים, כאשר מבני הנתונים מקושרים זה לזה באמצעות הרשימה. הפעולות על הרשימה כוללות בין השאר יצירת ראש הרשימה, הוספת איבר במקום נתון או בסוף הרשימה, הסרת איבר נתון, בדיקה האם הרשימה ריקה, גישה לרשומה המכילה איבר נתון, לולאת מעבר על איברים ברשימה.

הטווח הקצר – כל מתארי התהליכים המוכנים לריצה נגישים מתוך מבנה נתונים הקרוי runqueue. לכל מעבד יש runqueue משלו, כשכל runqueue מכיל מספר תורים של מתארי תהליכים, אחד לכל עדיפות של תהליך. כל תור ממומש כרשימה מעגלית כפולה.

הטווח הבינוני/ארוך – תהליך שצריך להמתין לאירוע כלשהו לפני המשך ריצתו נכנס לתור המתנה wait queue (הוא יוצא מה-runqueue ומוותר על המעבד). כל תור המתנה משויך לאירוע כלשהו כמו פסיקת חומרה, התפנות משאב מערכת לשימוש או אירועים אחרים כמו סיום תהליך. כאשר האירוע קורה, מערכת ההפעלה מעירה תהליכים מתוך התור.
תהליך ממתין בתור יכול להיות באחד משני מצבים:

- **בלעדי (exclusive)** – כאשר האירוע קורה, מעירים אחד מהתהליכים שמסומנים כבלעדי, כמו למשל כשהאירוע הוא שחרור של משאב שניתן לשימוש רק ע"י תהליך יחיד בו זמנית.
- **משותף (non-exclusive)** – כאשר האירוע המוערר קורה, מעירים את כל התהליכים שממתינים עם הסימון הזה.

זימון תהליכים

זימון טווח קצר – בוחר תהליך מתור המוכנים ומריץ אותו ב-CPU, מופעל לעיתים קרובות (מילי-שניות) וחייב להיות מהיר.
זימון טווח ארוך – בוחר איזה תהליך יובא לתור המוכנים, מופעל לעיתים רחוקות (שניות, דקות), יכול להיות איטי ותהליכים יכולים להשתחרר יחד מהמתנה בעקבות אירוע או אחד מקבוצה ביחס לאירוע מסוים.

מדדים להערכת אלגוריתם לזימון תהליכים לטווח קצר

- זמן תגובה (זמן המתנה + זמן ביצוע) מינימלי.
- תקורה (הזמן שמשקיעים שבו ה-CPU עסוק בניהול המשאבים של עצמו) מינימלית.

מטרות נוספות: ניצול מקסימלי של המעבד (יותר זמן שבו הוא פעיל) והספק/תפוקה מקסימלי (מס' התהליכים שמסתיימים בפרק זמן).

קיימים שני סוגים של תהליכים:

- **תהליך חישובי** – מעוניין להגדיל את זמן המעבד שלו ככל שניתן. הוא לא מוותר על זמן המעבד מרצונו, אלא הוא מופקע ממנו.
- **תהליך אינטראקטיבי** – מעוניין להקטין את זמן התגובה שלו ומוותר על המעבד מרצונו לאחר פרק זמן קצר. לרוב מדובר בקלט/פלט מול המשתמש.

First-Come, First-Served

התהליך שהגיע ראשון לתור הממתינים ירוץ ראשון, כך שלא מטפלים בתהליך הבא בתור עד שהקודם סיים (ללא הפקעות). האלגוריתם נותן עדיפות לתהליכים חישוביים, ממזער ניצול התקנים, המימוש שלו פשוט (תור FIFO) והוא לא מספק דרישות שיתוף. החיסרון המשמעותי הוא שיעילות המנגנון רגישה לסדר הופעת התהליכים, ויכולה להיות גדולה מאוד, כך שלמשל תהליך עתיר חישובים יכול לתפוס את המעבד ולמנוע מתהליכי קלט/פלט לרוץ.

Round Robin

תור מוכנים מעגלי. המעבד מוקצה לתהליך הראשון בתור ואם זמן הביצוע של תהליך גדל מקצבת זמן מסוימת q, התהליך מופסק ומועבר לסוף תור המוכנים (כלומר מתבצעת הפקעה).
אם q קטן, הזימון יהיה הוגן, כאילו כל תהליך רץ במעבד משלו בקצב של 1/N, כאשר N הוא מספר התהליכים. זמן התגובה יהיה כמעט ליניארי בזמן החישוב וב-N, אולם התקורה עלולה להיות גבוהה.
מצד שני, אם q יהיה גדול מאוד, RR יהפוך להיות כמו FCFS. זמן התגובה של RR הוא לכל היותר

פעמיים האופטימלי.
Selfish Round-Robin – תהליכים חדשים ימתינו בתור FIFO ואילו תהליכים ותיקים יוחזקו בתור RR לביצוע. כאשר אין תהליך בתור הוותיקים, נבחר את התהליך הראשון בתור החדשים ומצרפים אותו לוותיקים. בכל יחידת זמן, עדיפות התהליך גדלה עד שעוברת סף מסוים, והתהליך עובר לתור הוותיקים.

Shortest Job First
מריצים את התהליך עם זמן הביצוע המינימלי עד לסיומו. מתאפשר אם כל התהליכים מגיעים ביחד וזמן הביצוע של תהליך ידוע מראש. האלגוריתם הוא ללא הפקעות.

Shortest Remaining Time to Completion First
כאשר מגיע תהליך P_i שזמן הביצוע הנותר שלו קצר יותר מזמן הביצוע הנותר של התהליך שרץ כרגע P_k, נפקיע את המעבד וניתן ל-P_i לרוץ. האלגוריתם ממזער את זמן השהייה הממוצע במערכת, והוא אופטימלי כאשר תהליכים לא מגיעים יחד, בניגוד ל-SJF.

עדיפויות
כל תהליך מקבל עדיפות התחלתית, כאשר עדיפות התחלתית גבוהה ניתנת לתהליכים של משתמשים חשובים, לתהליכים שסיומם דחוף ולתהליכים אינטראקטיביים. התהליך עם העדיפות הגבוהה ביותר יקבל את המעבד (SJF לפי עדיפויות). הזימון יכול לגרום להרעבה של תהליכים עם עדיפות נמוכה, וניתן לפתור זאת אם הזדקנות (זמן שהייה ארוך) של תהליכים תגרום להגדלת העדיפות שלהם.
Multilevel Feedback Queues – קיימים מספר תורים לפי סדר עדיפות, כאשר תור גבוה מוקצה לתהליכים בעלי עדיפות גבוהה יותר והתורים הנמוכים מקבלים אחוז קטן יותר של זמן מעבד ו-quantum (הזמן שתהליך רץ ב-RR לפני שמפסיקים אותו) גדול יותר. כל תהליך מתחיל בתור הגבוה ביותר, ובסוף ה-quantum הוא יורד לתור נמוך יותר. הדבר יותר אפליה מתקנת של תהליכים עתירי קלט/פלט לעומת תהליכים עתירי חישוב (אם תהליך משחרר את המעבד לפני תום ה-quantum, הוא עולה לתור גבוה יותר). הדבר יותר עדיפות דינמית לפי השימוש במעבד, כך שככל שממתינים יותר, העדיפות גדלה וככל שזמן הריצה במעבד גדול יותר – העדיפות קטנה.

זימון ב-Unix
הזימון הוא לפי עדיפויות, כאשר עדיפות מספרית נמוכה טובה יותר. חישוב העדיפות מתבסס על דעיכה אקספוננציאלית, כל שתהליך שהשתמש לא מזמן במעבד מקבל עדיפות גבוהה יותר (גרועה), וככל שעובר הזמן, עדיפותו של התהליך דועכת והיא משתפרת. בצורה הזו, תהליכים שיוותרו מרצונם על המעבד (עתירי קלט/פלט), יחזרו אליו מהר יותר מתהליכים שעברו הפקעה (עתירי חישוב).

זימון ב-Windows NT
משתמשים ב-32 תורי עדיפויות, כאשר עדיפויות 16-31 נקראות Real time priority, עדיפויות 1-15 נקראות Variable priority ועדיפות 0 שמורה למערכת ההפעלה. מדיניות הזימון היא Round-Robin על התור העדיף ביותר שאינו ריק, כאשר העדיפות יורדת אם נצרך כל ה-quantum ועולה אם תהליך עובר מ-wait ל-ready. העדיפות נקבעת לפי סוג הקלט/פלט, כאשר קלט/פלט מהמקלדת מקנה עדיפות גבוהה.

זימון ב-Linux
גרסה נוספת של תורי עדיפויות. בדומה ל-Unix, גם כאן לכל תהליך יש עדיפות המורכבת מערך בסיס קבוע אותו המתכנת יכול לשנות, ובונוס דינמי שגדל כשתהליך חוזר מהמתנה וקטן (עד לערך שלילי כאשר התהליך נמצא הרבה בטווח הקצר. כל תהליך מקבל time slice שאורכו תלוי בעדיפות הבסיס, כך שתהליך עדיף יקבל time slice ארוך יותר. בהתאם לערך הנוכחי של עדיפותו הכוללת,

התהליך ישובץ לתור עדיפויות.
זמן המעבד מחולק לתקופות (epoch), כך שבכל תקופה, המערכת תיתן לכל התהליכים לרוץ החל מאלו שנמצאים בתור העדיפות הגבוה ביותר. כל תהליך רץ עד סיום ה-time slice שלו ואז מקבל time slice מעודכן עבור התקופה הבאה, ואז הוא עובר לרשימת ה-expired עד לתקופה הבאה. אין זמן מוגדר לתקופה והיא מסתיימת רק כאשר כל התהליכים ניצלו את ה-time slice שלהם.
יש הבדל בטיפול בין תהליכים חישוביים לאינטראקטיביים: בדרך כלל מחדשים לתהליכים אינטראקטיביים את ה-time slice לריצה נוספת בתקופה הנוכחית, בגלל שהם צורכים מעט זמן מעבד וזקוקים לו במיידי. תהליך יאופיין כאינטראקטיבי אם הוא ממתין הרבה זמן מיוזמתו (בטווח הנוכחי), כחלק מזמן הריצה שלו.

העדיפות נקבעת עבור כל תהליך במערכת בצורה הבאה: לכל תהליך יש עדיפות בסיסית שהיא 120. לכל תהליך מגדירים את הערך nice כך ש- $-20 \leq nice \leq +19$. העדיפות הסטטית של כל תהליך נקבעת לפי 120+nice.
כאשר תהליך חדש נוצר, תהליך האב מאבד מחצית מה-time slice לטובת תהליך הבן, זאת על מנת למנוע מצב שבו תהליך יכול להרוויח זמן מעבד בתוך אותה תקופה.

<u>זמן תהליכים scheduler</u> – רכיב תוכנה בגרעין של Linux שאחראי על זימון התהליך הבא למעבד וממומש באמצעות הפונקציה ()schedule. הוא מופעל בתגובה לפסיקת שעון כאשר תהליך סיים את ה-time slice שלו, בטיפול בקריאת מערכת שגורמת לתהליך הקורא לעבור להמתנה כך שהמעבד יכול להריץ תהליך אחר, בחזרה של תהליך מהמתנה על מנת לשבץ את התהליך מחדש ב-runqueue ולבדוק אם צריך לבצע החלפת הקשר או כאשר תהליך מחליט לוותר על המעבד מרצונו.

<u>Runqueue</u> – מבנה נתונים שמשמש את זמן התהליכים. כל runqueue מכיל את הנתונים הבאים: מספר התהליכים בתור, מצביע לתהליך שרץ כרגע, מצביע לתהליך ה-swapper, מצביע למערך תורי העדיפויות של התהליכים הפעילים שנותר להם זמן ריצה בתקופה הנוכחית, מצביע למערך תורי העדיפויות של התהליכים שמוכנים לרוץ אבל הם כילו את ה-time slice שלהם ואת הזמן שבו התהליך הראשון עבר מ-active ל-expired. מערך תורי העדיפויות מכיל כמה תהליכים נמצאים במערך, וקטור ביטים בגודל מספר דרגות העדיפות ומערך התורים עצמו.

<u>אפיון התנהגות תהליך</u> – Linux מודדת את זמן ההמתנה הממוצע של כל תהליך, כלומר סך כל זמן ההמתנה בטווח הבינוני פחות סך כל זמן הריצה. בכל פעם שתהליך מוותר על המעבד, ערך השעון נשמר בפונקציה ()schedule, וכאשר תהליך חוזר מהמתנה, זמן ההמתנה מתווסף לחשבון עד לגדול מקסימלי קבוע מראש MAX_SLEEP_AVG. כל פעימת שעון בה התהליך רץ מורידה מהממוצע עד למינימום 0. על פי שיטת חישוב זו תהליך עתיר חישוב צפוי להגיע לזמן המתנה ממוצע נמוך ואילו תהליך אינטראקטיבי צפוי להגיע לזמן המתנה ממוצע גבוה.

<u>חישוב דינמי של עדיפות תהליך</u> – Linux מעדכנת את העדיפות של כל תהליך "רגיל" בצורה דינמית בהתאם לזמן ההמתנה הממוצע של התהליך בעזרת הפונקציה ()effective_prio. הבונוס שניתן ומחושב בפונקציה מוגבל לגודל שבין 5- ל-5+ במטרה למנוע מצב שבו יתבצע היפוך עדיפויות בין תהליך שעדיפויותיהם הבסיסיות (הסטטיות) רחוק זו מזו. שיטת החישוב משפרת את העדיפות של תהליך שממתין הרבה ומרעה את העדיפות של תהליך שממתין מעט.

העדיפות ההתחלתית נקבעת ע"י המשתמש ולפי הערכתו אם התהליך אינטראקטיבי או חישובי. אם הערך nice הוא 20-, התהליך כמעט בוודאות אינטראקטיבי בעוד שאם nice=19, התהליך הוא חישובי. אם הערך שנקבע הוא 0, ה-scheduler מחליט בצורה דינמית מה התהליך יהיה, ואם העדיפות לא נקבעת, התהליך יקבל את העדיפות של אביו. תהליך יסווג כאינטראקטיבי אם העדיפות הדינמית שלו חורגת מהעדיפות הסטטית שלו כתוצאה מזמן המתנה ממוצע גבוה.
ככל שהעדיפות הסטטית משתפרת, קל יותר לתהליך להיות מסווג כאינטראקטיבי, כלומר הוא צריך

לצבוא זמן המתנה ממוצע נמוך בשביל להיחשב ככזה.
אם תהליך חישובי יוצא להמתנה ואז חוזר הוא יכול להיחשב כאינטראקטיבי לפרק זמן מסוים ואז לקבל פיצוי על ההמתנה. אם תהליך אינטראקטיבי מנצל time slice מלא הוא עלול לתקוע את האינטראקטיביים האחרים.

הרעבה – מתרחשת כאשר תהליך מונע מתהליכים אחרים לרוץ. כדי למנוע הרעבה של תהליכים חישוביים ע"י תהליכים אינטראקטיביים, מוגדר סף הרעבה על זמן ההמתנה של התהליכים ב-expired. אם יש תהליך ממתין והזמן שעבר מאז שהעברנו אותו להמתנה גדול מסף מסוים כפול מספר התהליכים שרצים ועוד 1, מפסיקים לתת time slices לתהליכים אינטראקטיביים בתקופה הנוכחית. הסף המוגדר פרופורציוני למספר התהליכים ב-runqueue, כך שככל שיש יותר תהליכים שמוכנים לרוץ, הסף יהיה גבוה יותר. המנגנון הוא גס, כי הוא מאפשר לאינטראקטיביים להרעיב את החישוביים במיוחד כשיש עומס על המערכת ולאחר שעוברים את הסף הוא מאפשר לחישוביים לעכב את האינטראקטיביים לזמן רב.

החלפת הקשר – Context Switch

פעולת החלפת ההקשר נדרשת בגלל שלכל תהליך יש הקשר ביצוע המכיל את כל המידע הדרוש לביצוע התהליך כמו מחסניות, רגיסטרים, דגלים, תכולת זיכרון וקבצים פתוחים. פעולת המיתוג שומרת את הקשר התהליך הנוכחי וטוענת את הקשר הביצוע של התהליך הבא למעבד.

ב-Linux לא ניתן לבצע החלפת הקשר כפויה לתהליך שנמצא ב-kernel mode, ואם בעקבות פסיקה נוצר צורך לבצע החלפת הקשר, היא תתבצע רק אחרי שהתהליך יסיים את הפעולה ב-kernel mode, לפני החזרה ל-user mode.
יש צורך לבצע context switch ב-3 מצבים: פסיקת שעון שבה מתגלה שנגמר ה-time slice של התהליך הנוכחי, פסיקה שבה משתחרר מהמתנה תהליך בעל עדיפות טובה יותר מהתהליכים שמוכנים לריצה כולל התהליך שרץ כרגע או ויתור של תהליך על המעבד מרצונו וקורא ל-scheduler.

הפונקציה Scheduler_tick() – פונקציה המופעלת בכל פסיקת שעון ומעדכנת את נתוני הזימון של התהליכים. היא מופעלת כאשר הפסיקות חסומות כדי למנוע שיבוש נתונים ע"י הפעלת הפונקציה במקביל, מזהה צורך בהחלפת הקשר בעקבות סיום time slice של התהליך הנוכחי ומשתמשת בפונקציות המתאימות להוצאה והכנסה של תהליכים לאחד ממערכי התורים. הפונקציה שומרת שהמערכת לא תורעב ע"י הוצאת תהליך אינטראקטיבי להמתנה (אחרת היא תיתן לו time slice נוסף).

ביצוע החלפת הקשר – schedule() היא הפונקציה היחידה שמפעילה את החלפת ההקשר והיא זו שבוחרת איזה תהליך יזומן למעבד. היא קוראת לפונקציה context_switch() שמבצעת את ההחלפה, ושתיהן מתבצעות כשהפסיקות חסומות על מנת למנוע הפעלה רקורסיבית של הזמן והחלפת ההקשר ובכך לשבש את פעולת מערכת ההפעלה.
החלפת ההקשר היא בקירוב רצף הפעולות הבא:

1. שמירת נתוני הקשר התהליך הנוכחי.
2. מעבר למחסנית הגרעין של התהליך הבא (התהליך הנוכחי החדש).
3. טעינת נתוני הקשר התהליך הנוכחי החדש.
4. קפיצה לכתובת הבאה לביצוע של התהליך הנוכחי החדש.

מרבית ההקשר של תהליך ב-Linux מאוחסן במתאר התהליך ולכן אין צורך לשמור ולטעון אותו מחדש בכל החלפת הקשר, אלא רק את הרגיסטרים והדגלים. פעולת החלפת ההקשר כוללת גם החלפת איזורי הזיכרון אליהם המעבד ניגש ב-user mode.

Task State Segment – איזור TSS הוא איזור זיכרון בגרעין המכיל מידע מתוך הקשר התהליך הנוכחי המתבצע במעבד. לכל מעבד יש TSS משלו מתוך אילוץ של החומרה. המעבד קורא את כתובת מחסנית הגרעין של התהליך הנוכחי משדה ב-TSS בעת מעבר ל-kernel mode. כשמתבצעת החלפת הקשר, מעדכנים את השדות המתאימים ב-TSS.

הפונקציה **switch_to()** – פונקציה שמשלימה את רצף הפעולות במסגרת החלפת ההקשר. היא מוגדרת באופן מיוחד, כך שהיא מקבלת פרמטרים ברגיסטרים ולא במחסנית. היא מופעלת באמצעות קפיצה מהמאקרו switch_to שמופיע בפונקציה context_switch(). היא כתובה בשפת C והיא ספציפית למעבד עליה היא רצה.

יצירת תהליך חדש – קריאות המערכת המשמשות ליצירת תהליכים חדשים משתמשות בפונקציה פנימית של הגרעין שנקראת do_fork() לבניית ההקשר של התהליך החדש. הפונקציה do_fork() מעתיקה את מרבית הנתונים ממתאר תהליך האב למתאר חדש של תהליך הבן שהיא יוצרת. היא בונה מחסנית גרעין לתהליך הבן ע"י קריאה לפונקציה copy_thread(), קישור מתאר תהליך הבן לרשימת התהליכים ולבני משפחתו, מקשרת בין ה-PID של תהליך הבן למתאר תהליך הבן, מעבירה את תהליך למצב של TASK_RUNNING ומכניסה אותו ל-runqueue. לסיום, הפונקציה מחזירה את ה-PID של תהליך הבן והערך הזה מוחזר לתהליך האב.
copy_thread() – מאתחלת את תכולת מחסנית הגרעין של תהליך הבן ואת שדה threads במתאר תהליך הבן.
ret_from_fork() – פונקציה שמופעלת כשתהליך הבן מזומן לראשונה למעבד לאחר הקשר. ביצוע הקוד בה יגרום לסיום הקריאה fork() בתהליך הבן עם ערך מוחזר 0.

סיום ביצוע תהליך – תהליך מסיים את ביצוע הקוד ע"י קיראת המערכת exit(). אם הקוד לא קורא לה במפורש, מתבצעת קריאה אוטומטית ל- exit() אחרי החזר מ-main(). ביצוע הקוד יכול להיקטע בעקבות אירועים נוספים כמו תקלה לא מטופלת במהלך ביצוע הקוד כמו גישה לא חוקית לזיכרון, או אם תהליך אחד נהרג ע"י תהליך אחר.
הפונקציה do_exit() מופעלת בכל מקרה של סיום ביצוע תהליך. היא משחררת את המשאבים שבשימוש התהליך, מעדכנת את השדה exit_code במתאר התהליך שיכיל את הערך שמוחזר ע"י exit(), מעדכנת את קשרי המשפחה כך שכל הבנים של התהליך שסיים יהפכו לבנים של init, מעבירה את מצבו של התהליך ל-TASK_ZOMBIE וקוראת ל-schedule() שיוצא את התהליך מ-runqueue ותזמן תהליך אחר לביצוע במעבד.
מתאר התהליך מפונה רק כאשר תהליך האב מקבל חיווי על סיום התהליך והוא מתבצע ע"י הפונקציה release_task(). הפונקציה הזו מנתקת את התהליך מרשימת התהליכים וממנגנון הקשיור ל-PID, ומפנה את השטח המוקצה למתאר התהליך ומחסנית הגרעין.

חוטים threads

תהליכים דורשים משאבי מערכת רבים והמעבר בין תהליכים (החלפת הקשר) לוקח זמן רב, וגורמת לפגיעה ביעילות המחשב ככל שהיא מתבצעת פעמים רבות יותר. אולם ריבוי תהליכים נדרש לפתרון של הרבה בעיות בצורה פשוטה וקלה יותר. התהליכים שנוצרים לטיפול בבקשות דומים זה לזה כי יש להם את אותו קוד ואותם משאבים ונתונים, אולם הם מטפלים בבקשות שונות ויכולים להימצא בשלבים שונים במהלך הטיפול. הפתרון שהוצא הוא שימוש בחוטים.

חוטים – יחידת ביצוע (בקרה) בתוך תהליך. במערכות הפעלה מודרניות תהליך הוא רק מיכל לחוטים, כאשר לכל חוט דרושים program counter, מחסנית ורגיסטרים משלו. המחסנית וה-PC של כל חוט נמצא בכל מהמקומות המתאימים במרחב הכתובות של תהליך מרובה חוטים.
יש יתרונות רבים לשימוש בחוטים: יצירת חוט יעילה יותר היות ויוצרים רק thread control block

ומקצים מחסנית והחלפת ההקשר בין חוטים של אותו תהליך מהירה יותר, מערכת מרובת חוטים מנצלת טוב יותר את המשאבים, כך שאם למשל חוט אחד נחסם, חוטים אחרים של אותו תהליך ממשיכים לרוץ, התקשורת בין חוטים ששייכים לאותו תהליך נוחה יותר בגלל הזיכרון המשותף, וניתן באמצעות המערכת לכתוב תכנות מובנה יותר.
החיסרון העיקרי הוא שאין הגנה בין חוטים באותו תהליך, כך שחוט עלול לדרוס את המחסנית של חוט אחר ויכולה להיות גישה לא מתואמת למשתנים גלובליים.

חוטי משתמש – מוגדרים ע"י סביבת התכנות. החוטים האלו לא דורשים קריאות מערכת, הזימון שלהם נעשה בשיתוף פעולה ואין החלפת הקשר בתוך הגרעין. כשאחד החוטים נחסם קיים מנגנון שמחכה עד שהחסימה תשתחרר ועד אז חוט אחר רץ במקומו.
חוטי מערכת – חוטים המוכרים למערכת ההפעלה והיא אחראית לניהול שלהם.

Thread Pooling

אם נניח שזמן הריצה הממוצע שכל חוט מקבל בשנייה הוא c וזמן המעבד הממוצע שכל תהליך מקבל בשנייה הוא p, נסיק כי אין טעם לייצר מ-p/c חוטים. כמות כזו מבטיחה ניצול מלא של זמן המעבד המוקצה לתהליך, תוספת חוטים רק תגדיל את התקורה של החלפת ההקשר ואת הזמן המבוזבז על פעולת סנכרון, ואם נגיע למצב של עודף חוטים, נגרום לכך ששטח המחסנית של כל חוט ייקטן.
ב-thread pooling אנו מייצרים רק כמות סופית של חוטים. החוט הראשי הינו המנהל ובמקום ליצור חוט עבודה עבור כל בקשה, הוא יאכסן את הבקשות המגיעות בתור FIFO. כל חוט אחר הינו חוט עובד והוא מוציא מהתור משימה ומבצע אותה.
גודל ה-pool הינו דינמי, כאשר אם הגודל לא מספיק, כלומר אורך תור המשימות גדל והתהליך לא מספיק לנצל את כל ה- quantumשלו, אז מגדילים ב-k נוספים עד לסף עליון קבוע. באותו אופן, אם במשך הרבה זמן יש חוטים מובטלים, מבטלים k מהם.

חוטים ב-Linux

Linux תומכת בחוטים ברמת גרעין מערכת ההפעלה. חוטי המערכת הם למעשה תהליכים רגילים המשתפים ביניהם משאבים כמו זיכרון, גישה לקבצים וחומרה. לכל חוט, בהיותו תהליך רגיל, יש מתאר תהליך משלו ו-PID משלו. בהתאם לתקן POSIX, המתכנת מצפה שלכל החוטים השייכים לאותו תהליך ניתן יהיה דרך PID יחיד של התהליך שמכיל אותם. פעולות על ה-PID של התהליך צריכות להשפיע על כל החוטים בתהליך.
כדי לאפשר את ההתייחסות לכל החוטים באותו תהליך, מוגדר בגרעין של Linux העיקרון של קבוצת חוטים (thread group): כל החוטים השייכים לאותו תהליך יימצאו בקבוצה אחת, מתארי התהליכים של כל החוטים באותה קבוצה יהיו מקושרים באמצעות שדה thread_group במתאר התהליך, שדה tgid במתאר התהליך יכיל את ה-PID המשותף לכל החוטים באותה קבוצה (זהו ה-PID של החוט הראשון של התהליך) ולכן פעולות על ה-PID המשותף מתורגמות לפעולה על קבוצת החוטים המתאימה ל-PID ואם לחוט כלשהו יש בנים (תהליכים), הם הופכים להיות בנים של חוט אחר בקבוצת האב לאחר מותו.

קריאת המערכת clone() – מאפשרת לתהליך ליצור תהליך נוסף המשתף איתו משאבים ונתונים לפי בחירה. קריאת מערכת זו היא הבסיס לספריות התמיכה בחוטים מתוך user mode. הדגלים שהיא מקבלת קובעים את צורת השיתוף בין התהליך הקורא והתהליך החדש

POSIX Threads API

- יצירת חוט - **pthread_create()** יוצרת חוט חדש המתבצע במקביל לחוט הקורא בתוך אותו תהליך. החוט החדש מתחיל לבצע את הפונקציה המופיעה בפרמטר start_routine ונהרג

בסיום ביצוע הפונקציה. במקרה של הצלחה היא מחזירה 0 ומזהה החוט החדש מוכנס למקום המוצבע ע"י thread.

- סיום חוט - **pthread_exit()** גורמת לחוט הקורא לסיים את פעולתו וערך הסיום מוחזר לחוט שממתין לסיום חוט זה. סיום פעולת החוט הראשי לא מסיים את כל החוטים בתהליך.
- קבלת מזהה החוט - **pthread_self()** מחזירה את המזהה של החוט שקרא לה.
- המתנה לסיום חוט - **pthread_join()** – החוט הקורא ממתין לסיום החוט המזוהה ע"י הפרמטר th. ניתן להמתין על סיום אותו חוט פעם אחת לכל היותר, אחרת הפעולה תיכשל. כל חוט יכול להמתין לסיום כל חוט אחר באותו תהליך.
- הריגת חוט - **pthread_cancel()** מסיימת את ביצוע החוט המזוהה ע"י הפרמטר thread.

ביצוע fork() בתוך חוט – כאשר חוט קורא ל-fork(), נוצר תהליך חדש שהוא הבן של החוט הקורא בלבד. חוט אחר בקבוצה של החוט הקורא לא יכול לבצע wait() לתהליך הבן שנוצר. גם לתהליך הבן החדש יש חוטים משלו, כאשר בהתחלה יש רק חוט יחיד (החוט הראשי) ולאחר מכן ניתן ליצור בו חוטים נוספים. אם חוט האב מבצע wait() מחכים שכל תהליך הבן יסתיים, ואם חוט האב מת, בניו עוברים לאחד מאחיו (באופן רנדומלי).

ביצוע execv() בתוך חוט – אם קריאה ל-execv() מצליחה, החוט הקורא מתחיל מחדש בתור חוט ראשי בקבוצה חדשה של תהליך חדש, כולל הקצאת משאבים מחדש. כל שאר החוטים האחרים מופסקים.

סיום ביצוע תהליך – אם חוט כלשהו מתוך תהליך קורא ל-exit(), הסתיים ביצוע הקוד של החוט הראשי או שביצוע אחד החוטים גורם לתקלה לא מטופלת, מתבצע סיום ביצוע התהליך כולו, כאשר כל החוטים בקבוצה.

אמצעי סנכרון ותיאום בין תהליכים

מסלולי בקרה

גרעין מערכת ההפעלה מטפל בבקשות מסוגים שונים ע"י ביצוע סדרת פקודות (פעולות). סדרת הפקודות המתבצעת מהרגע שמתקבלת בקשה ועד סיום הטיפול בה נקראת מסלול בקרה בגרעין. המעבד יכול למתג בין מספר מסלולי בקרה בתוך הגרעין, למשל בגלל ביצוע החלפת הקשר בגרעין, או ביצוע פסיקה נוספת בזמן טיפול בפסיקה אחרת. מבני הנתונים בגרעין עלולים להשתבש כתוצאה מגישה בלתי מתואמת ממספר מסלולי בקרה, ולכן צריך להגן עליהם ע"י סנכרון.
קריאת מערכת לא חוסמת מתבצעת בצורה אטומית ביחס לקריאות מערכת אחרות, כך שאם היא לא משתמשת במבני נתונים הנגישים ממסלולי בקרה של פסיקות אחרות, היא יכולה לגשת לנתונים בבטחה. קריאת מערכת חוסמת (מוותרת על המעבד) חייבת להשאיר את מבני הנתונים בגרעין תקינים לפני הוויתור על המעבד, ובחזרה לביצוע היא צריכה לבדוק שהנתונים לא השתנו.

תיאום בין תהליכים

התהליכים משתפים פעולה ע"י גישה למשאבים משותפים או אם מעבירים נתונים מתהליך אחד לשני דרך התקן משותף. אי תיאום בין שני תהליכים שמבצעים את אותו קטע קוד יכולה ליצור מצב של race condition, שבו שני חוטים מנסים להגיע למשאב מסוים בלי שום תיאום ביניהם, וגורמים לתוצאת ריצה לא צפויה. יש צורך מנגנון לשליטה בגישות מקביליות למשאבים משותפים כדי שיהיה ניתן לחזות באופן דטרמיניסטי את התוצאות.

קטע קריטי – הקוד שניגש למשאב המשותף מכונה קטע קריטי, כאשר לא בהכרח מדובר על אותו קוד לכל החוטים. מוסכמה תכנותית היא לעטוף קטע קריטי בקוד כניסה וקוד יציאה.

תכונות רצויות

- מניעה הדדית – חוטים לא יבצעו בו זמנית את הקטע הקריטי, כאשר חוטים מעוניינים יחכו בקטע הכניסה, וכאשר החוט הנוכחי יוצא מהקטע הקריטי, הם יוכלו להיכנס.
- התקדמות – אם יש חוטים שרוצים לבצע את הקטע הקריטי, חוט כלשהו יצליח להיכנס כל עוד אין חוט אחר בתוך הקטע הקריטי (אין קיפאון).
- הוגנות – אם יש חוט שרוצה לבצע את הקטע הקריטי, הוא יצליח, ורצוי תוך מספר צעדים חסום (אין הרעבה).

סוגי אמצעי סנכרון

- הוראות אטומיות – פעולות עדכון נתונים המבוצעות באופן אטומי ברמת המכונה ביחס לכל המעבדים. הקטע הקריטי מוכל בהוראה האטומית, שמנצלת תמיכה של החומרה.
- חסימת פסיקות – חסימת הפסיקות במעבד בו מתבצע מסלול הבקרה. זו אחד האמצעים החשובים להגן על הקטע הקריטי והוא מאפשר למסלול בקרה להתקדם ללא חשש מקטיעה ע"י מסלול בקרה של פסיקות אחרות. מצד שני, חסימת הפסיקות לזמן רב עלולה לפגוע בביצועים ולאובדן של פסיקות חיוניות. בסיום הקטע הקריטי משחזרים את ערך דגל הפסיקות מלפני כיבויו.
- סמפורים כלליים – סוג של מנעול המיועד לשימוש פנימי בגרעין בלבד. המתנה של מסלול בקרה לסמפור גורמת להמתנת התהליך שרץ כרגע.

מנעולים – אבסטרקציה שמבטיחה גישה בלעדית למידע בעזרת שתי פונקציות – נעילה ושחרור, שמופיעות בזוגות. רק חוט אחד מחזיק את המנעול, ורק הוא יכול לבצע את הקטע הקריטי. חסימת פסיקות מונעת החלפת חוטים ומבטיחה פעולה אטומית על המנעול. מאפשרים פסיקות בתוך הלולאה אחרת לא נוכל לצאת מהלולאה של הפסיקה שרצה באותו רגע.

Spinlocks – מימוש של מנעול באמצעות busy waiting: אם המנעול תפוס (ע"י גישה למשתנה), בודקים שוב. המימוש מאוד בזבזני, כי חוט שבודק אם המנעול תפוס מבזבז זמן cpu, ואילו החוט שמחזיק במנעול לא יכול להתקדם. קיים רק במערכת מרובת מעבדים, ורק בו הוא המתנה יעילה כשמדובר בנעילות בקצרות מאוד כפי שקורה בגרעין, מפני שאין את התקורה של כניסה ויציאה מהמתנה.

סמפור semaphore – מנגנון תיאום עילאי המורכב משני שדות: ערך מספרי שלם ותור של חוטים/תהליכים ממתינים. קיימות 2 פעולות על סמפור:

- **Wait** – מקטין את ערך המונה ב-1 וממתין עד שערכו של הסמפור אינו שלילי.
- **Signal** – מגדיל את ערך המונה ב-1 ומשחרר את אחד הממתינים.

שניהם נקראים ע"י המשתמש, ואילו מערכת ההפעלה דואגת שהכל יתבצע. קיימים שני סוגים עיקריים של סמפורים:

1. **סמפור בינארי** – ערכו ההתחלתי הוא 1 וזה גם הערך המקסימלי. הוא מאפשר גישה בלעדית למשאב (בדומה למנעול).
2. **סמפור מונה** – ערכו ההתחלתי הוא N>0 והוא שולט על משאב עם N עותקים זהים. חוט יכול לעבור ()wait בסמפור כל עוד יש עותק פנוי של המשאב.

בעזרת השימוש בסמפורים נוכל למנוע מחוט לקרוא או לכתוב נתונים למקום מסוים בזיכרון אם מישהו כותב לאותו מקום. לסמפורים יש מספר חסרונות: הם לא מפרידים נעליה, המתנה או ניהול המשאבים.

משתני תנאי – מימוש מודרני יותר של מנגנון תיאום ונעילה. הפעולות על משתני תנאי הם:

- **Wait** – שחרור המנעול (חייב להחזיק בו), המתנה לפעולת signal והמתנה למנעול (כשחוזר מחזיק במנעול).
- **Signal** – מעירה את אחד הממתינים ל-cond (משתנה התנאי) שעובר להמתין למנעול. אם אין ממתינים, אין לו השפעה (הולך לאיבוד). התהליך לא מקבל את המנעול באופן אוטומטי אלא מחכה עד שהוא יתפנה ואז תופס אותו. הדבר מאפשר שליטה על החוט/תהליך הבא שאמור לתפוס את המנעול.
- **Broadcast** – מעיר את כל התהליכים הממתינים שעוברים להמתין למנעול (עד שאחד מהם תופס אותו). הולך לאיבוד אם אין ממתינים.

מנגנוני תיאום ב-Windows NT – כל רכיב במערכת ההפעלה הוא אובייקט תיאום שיכול להמתין ו/או לשחרר. קיימים מספר אובייקטים מיוחדים: mutex (מנעול הוגן), event (משתנה תנאי), semaphore (סמפור מונה, לא הוגן) ו-critical section (mutex קל משקל המיועד לחוטים באותו תהליך).

סינכרוניזציה ב-Linux

הסנכרון בין חוטי POSIX ב-Linux נעשה בעזרת מספר מנגונים: mutexes, סמפורים (רק עובר חוטים) ומשתני תנאי.

מנעולי mutex

מאפשרים לחוט אחד בדיוק להחזיק בהם, כלומר לנעול אותם. כל חוט אחר שיבקש להחזיק במנעול ייחסם עד שהחוט ישתחרר, ורק החוט שמחזיק במנעול אמור לשחרר אותו. הם משמשים בדרך כלל להגנה על גישה לנתונים משותפים בתוך קטע קוד קריטי, ע"י נעילת המנעול בכניסה לקטע הקריטי ושחרורו בסופו.

פעולות על מנעולי mutex –

- אתחול המנעול.
- פינוי המנעול בתום השימוש (נכשל אם המנעול מאותחל אבל נעול).
- נעילת המנעול. הפעולה חוסמת עד שהמנעול מתפנה ואז נועלת אותו (אם המנעול כבר נעול, התהליך יוצא להמתנה עד שהמנעול מתפנה).
- ניסיון לנעילת המנעול (trylock). נכשל אם המנעול כבר נעול, אחרת נועלת אותו.
- שחרור.

סוגים של מנעולי mutex –

- **Mutex מהיר** – מהיר אבל לא בודק שגיאות. אם מנסים לנעול שוב את החוט המחזיק במנעול מקבלים מצב של קיפאון, מותר לשחרור מנעול ע"י חוט שלא מחזיק במנעול.
- **Mutex רקורסיבי** – משתמש במונה למספר הנעילות. נעילה חוזרת תגדיל את מונה הנעילה העצמית ב-1, ושחרור יקטין אותו. המנעול יישאר נעול כל עוד המונה גדול מאפס.
- **Mutex בודק שגיאות** – לא מאפשר לנעול שוב את החוט שמחזיק במנעול או לשחרר מנעול ע"י חוט שלא מחזיק בו. המנעול שמומלץ להשתמש בו.

סמפורים כלליים

לכל סמפור יש מונה, תור ומנעול שתפקידו להגן על הקוד של הפעולות של הסמפור. קיימות שתי פעולות בסיסיות על הסמפור: wait שמקטינה את המונה ב-1 אם הוא גדול מ-0, ו-post שמוציאה ומעירה את החוט הראשון בתור אם תור הממתינים ריק, אחרת היא מגדילה את המונה ב-1. תור הממתינים הוא מסוג FIFO לפי עדיפויות (קודם ממיינים לפי עדיפות ואחר כך FIFO).
אפשר להשתמש בסמפור בשביל לקבוע סדר ריצת תהליכים (למשל תהליך שירוץ רק אחרי שכמה תהליכים שונים סיימו).

משתני תנאי

מנעול שמחכה לאירוע ספציפי, ומאפשר לחוט להמתין לאירוע כלשהו, כפי שמוגדר ע"י המתכנת.

פעולות על משתני תנאי –

- אתחול משתנה תנאי לפני השימוש.
- פינוי משתנה תנאי בסיום השימוש. הפעולה תיכשל אם יש חוטים הממתינים למשתנה.
- המתנה על משתנה תנאי. החוט הממתין חייב להחזיק במנעול mutex לפני הקריאה.
- שחרור חוט ממתין על משתנה תנאי. אם אין חוט שממתין כרגע על משתנה התנאי, הפעולה חסרת השפעה. אם קיימים חוטים ממתינים על משתנה התנאי, אחד החוטים הממתינים (אין הגינות מובטחת) מפסיק להמתין עליו ועובר להמתין על ה-mutex ויחזור לפעילות אחרי שהוא יתפוס וינעל את המנעול.
- שחרור כל החוטים הממתינים על משתנה התנאי. כל החוטים שממתינים למשתנה מפסיקים להמתין ועוברים להמתין ל-mutex. הם יחזרו לפעילות בזה אחר זה לאחר שינעלו את המנעול (הסדר לאו דווקא הוגן).

הדרך הפשוטה ביותר לקשר בין אירוע המציין קיום מצב רצוי כלשהו למשתנה תנאי היא להוסיף תנאי (if) לפני ההמתנה לאירוע, ואילו יוצר האירוע יוסיף תנאי שאם יתקיים, הוא יאותת על שחרור המשתנה לממתינים. ייתכן שלפני שהחוט הממתין ינעל את ה-mutex מחדש יספיק חוט אחר להיכנס ולשנות את הנתונים כך שהמצב הרצוי כבר לא מתקיים, ולכן השימוש הנכון הוא בלולאת while כשבודקים אם התנאי התקיים.

מנעול קוראים-כותבים

משאב עם שתי פעולות: קריאה וכתיבה. מספר חוטים קוראים יכולים לגשת למשאב בו זמנית, אולם חוט כותב צריך גישה בלעדית למשאב. המימוש העיקרי של המנעול הוא באמצעות מוניטור (mutex עם 0 או יותר משתני תנאי).

קיפאון

קבוצת תהליכים/חוטים שבה כל אחד ממתין למשאב המוחזק ע"י מישהו אחר בקבוצה.

מתקיימים התנאים הבאים:

1. יש מניעה הדדית – משאבים שרק תהליך אחד יכול לעשות שימוש בהם בו זמנית, כאשר כמה עותקים של אותו משאב נחשבים למשאבים נפרדים. משאבים שאין צורך במניעה הדדית עבורם אינם יכולים לגרום לקיפאון.
2. החזק והמתן – תהליך מחזיק משאב ומחכה למשאב אחר שבשימוש אצל תהליך אחר.
3. לא ניתן להפקיע משאבים
4. המתנה מעגלית

דרכי התמודדות עם קיפאון

- מניעה – קובעים מדיניות שאמורה למנוע מקיפאון להתרחש. למשל מניעת המתנה מעגלית ע"י קביעת סדר מלא בין המשאבים, קביעה שתהליכים יבקשו משאבים רק בסדר עולה או שתהליך שמבקש משאב מסדר נמוך יהיה חייב לשחרר קודם משאבים מסדר גבוה.
- גילוי קיפאון – משתמשים באלגוריתם שאמור לזהות מעגל (למשל כל תהליך הוא צומת בגרף והיחס בין התהליכים הוא הקשתות בין התהליכים, והאלגוריתם מזהה מעגל בגרף).
- היחלצות מקיפאון – ביטול כל התהליכים שנמצאים במצב קיפאון, ביטול תהליכים אחד-אחד עד להיחלצות או הפקעת משאבים לצורך סיום מצב הקיפאון.
- התחמקות מקיפאון – דואגים להשאיר את המערכת במצב בטוח שבו תהליכים מצהירים על כוונתם לבקש משאבים ואסור לתהליך לבקש משאב שהוא לא הצהיר עליו. המצב יהיה

בטוח אם קיימת סדרת הקצאות לכל המשאבים שהוצהרו מבלי להיכנס לקיפאון. אם היענות לבקשה עלולה להכניס את המערכת למצב לא בטוח, הבקשה נדחית.

רוב מערכות ההפעלה לא דואגות למנוע קיפאון, היות ומדובר בפעולה כבדה. האחריות לכך נותרת אצל המתכנת, שיכול להרוג תהליך כדי להיחלץ מקיפאון, אבל זה עלול לגרום למערכת להיות במצב לא תקין.

שיטת Watchdog לגילוי והיחלצות מקיפאון – ממומשת בחומרה ע"י רגיסטר מיוחד שסופר אחורה ערך מונה וכשהמונה מגיע לאפס הוא מבצע reset להתקן (מכבה ומדליק אותו מחדש). החוטים במערכת מדי פעם ניגשים לרגיסטר ומעדכנים את המונה לערך התחלתי גבוה. אם יש קיפאון, החוטים לא יעדכנו את המונה ויתבצע reset.
שיטה זו מחייבת שיתוף פעולה של החוטים העובדים ואם רוצים לוודא שאף חוט לא מעורב בשום קיפאון, יש להשתמש ברגיסטר נפרד לגל חוט.

קלט/פלט ותקשורת תהליכים ב-Linux

ב-Linux פעולות הקלט/פלט של תהליך מול התקן מבוצעות באמצעות descriptors, כאשר כל אחד מהם הוא אינדקס של כניסה בטבלת ה-Process Descriptors Table (PDT) שמכילה 256 כניסות. כל כניסה בטבלה מצביעה על אובייקט ניהול מטעם התהליך עבור ההתקן המקושר ל-descriptor (כל שורה יכולה להיות פעילה או להכיל NULL), כאשר הטבלה עצמה מוצבעת ממתאר התהליך. כל התקני הקלט/פלט האפשריים מופעלים באותו האופן באמצעות ממשק תקשורת אחיד דרך ה-descriptor.

קיימים 3 ערכי descriptors המהווים ברירת מחדל בקישור להתקנים הבאים:

- **0 (stdin)** – מקושר לקלט הסטנדרטי (לרוב המקלדת).
- **1 (stdout)** – מקושר לפלט הסטנדרטי (לרוב תצוגת טקסט במסוף).
- **2 (stderr)** – מקושר לפלט השגיאות הסטנדרטי, בדרך כלל גם הוא תצוגת טקסט במסוף.

ניתן לשנות את קישור ה-descriptors להתקנים באופן דינמי.

קריאות מערכת בסיסיות

- פתיחת התקן לגישה **open()** – ההתקן המבוקש נפתח לגישה לפי התכונות המועברות בדגלים ולפי ההרשאות המוגדרות. הקריאה בוחרת את התא הריק בעל הכניסה הנמוכה ביותר בטבלה. הדגלים מציינים אם הקובץ נפתח לקריאה ו/או כתיבה, ותכונות אופציונאליות נוספות.
- סגירת התקן לגישה **close()** – סוגר את ה-descriptor כך שלא יהיה ניתן לגשת אליו.
- קריאת נתונים מההתקן **read()** – מנסה לקרוא עד count (פרמטר) בתים מתוך ההתקן המקושר לתוך חוצץ שמסופק. מחזירה, אם הצליחה, את מספר הבתים שנקראו בפועל מההתקן (כולל 0), אחרת מחזירה 1-. ייתכן שייקראו פחות מ-count בתים וייתכן שלא ייקראו בכלל. מחוון הקבצים מקודם בכמות הבתים שנקראו בפועל מההתקן, כך שבפעולת הגישה הבאה לקובץ ניגש לנתונים שאחרי הנתונים שנקראו בפועל הנוכחית. הפעולה תחסום את התהליך הקורא (יועבר להמתנה עד שיהיו נתונים זמינים לקריאה בהתקן.
- כתיבת נתונים להתקן **write()** – מנסה לכתוב עד count בתים מתוך חוצץ מסופק להתקן המקושר, ומחזירה את מספר הבתים שנכתבו בפועל (גם אם לא נכתבו בתים). אם נכשלת, מחזירה 1-. בדומה ל-()read, מחוון הקבצים מקודם לפי כמות הבתים שנכתבו בפועל, וגם היא חוסמת את התהליך בפעולה על התקנים מסוימים.

שיתוף קלט/פלט בין חוטים ותהליכים

טבלת ה-descriptors הינה גלובלית לתהליך ולכן משותפת לכל החוטים שבו. לכל תהליך יש PDT משלו, כאשר פעולת ()fork יוצרת עותק נוסף של הטבלה אצל תהליך הבן, כך שהבן יכול לגשת לאותם התקנים כמו אביו וה-descriptors ששוכפלו הינם שותפים, כלומר מצביעים לאותו אובייקט ניהול, ובפרט חולקים את אותו מחוון קובץ.
תהליכים וחוטים שמשתמשים ב-descriptors שותפים או משותפים צריכים לטפל את פעולות הגישה להתקן על מנת שלא לשבש זה את פעולת זה. פעולת ()execv לא משנה את טבלת ה- descriptors של התהליך, למרות שהוא מאותחל מחדש.

מנגנוני תקשורת בין תהליכים

Pipes

ערוצי תקשורת חד-כיווניים המאפשרים העברת נתונים לפי סדר FIFO, כאשר הם גם משמשים גם לסנכרון תהליכים. המימוש שלהם ב-Linux הוא כאובייקטים של מערכת הקבצים, למרות שהם אינם צורכים שטח דיסק כלל ואינם מופיעים בהיררכיה של מערכת הקבצים. כדי ליצור pipe משתמשים בקריאת מערכת מיוחדת והגישה אליו נעשית באמצעות שני descriptors (לקריאה ולכתיבה).
ה-pipe שנוצר הינו פרטי לתהליך ולא נגיש לתהליכים אחרים במערכת. הדרך היחידה לשתף pipe בין תהליכים שונים היא באמצעות קשרי משפחה, כאשר תהליך אב יוצר pipe ואחריו יוצר תהליך בן באמצעות ()fork, ואז לאב ולבן יש גישה ל-pipe באמצעות ה-descriptors שלו המצויים בשניהם. לאחר שסוגרים את ה-descriptors, המשאבים של ה-pipe מפונים באופן אוטומטי.

<u>קריאה וכתיבה ל-pipe</u> – הפעולות מתבצעות באמצעות ()read ו-()write על ה-descriptors של ה-pipe. בדרך כלל, תהליך מבצע רק אחד מהפעולות ונהוג לסגור את ה-descriptor השני שלא בשימוש. יש צורך לתאם בין הקוראים בגלל שה-descriptors של הקריאה בהם משותפים (כנ"ל לגבי הכותבים).
קריאה מ-pipe תחזיר את כמות הנתונים המבוקשת אם היא נמצאת ב-pipe או פחות, אם זו הכמות הזמינה באותו הרגע (אפס יוחזר אם כל ה-write descriptors סגורים וה-pipe ריק). אם יש כותבים ל-pipe והוא ריק, התהליך יחסם, וכאשר תתבצע כתיבה, הנתונים שנכתבו יוחזרו עד לכמות המבוקשת.
כתיבה ל-pipe תבצע כתיבה של כל הכמות המבוקשת אם יש מספיק מקום פנוי ב-pipe ואם יש קוראים (read descriptors) ואין מספיק מקום ב-pipe, הכתיבה תחסום את התהליך עד שניתן יהיה לכתוב את כל הכמות הדרושה. ה-pipe מוגבל בגודלו (כ-4K) ולכן כתיבה ל-pipe שאין בו מספיק מקום פנוי ושאין עבורו קוראים (read descriptors פתוחים) תיכשל.

<u>הכוונת קלט ופלט</u> – אחד השימושים הנפוצים ב-descriptors בכלל וב-pipe בפרט הינו הכוונת הקלט והפלט של תוכניות. כאשר רוצים שהקלט יבוא מתוך (או שהפלט יישלח אל) תהליך אחר, מחליפים את התקן הקלט או הפלט ב-pipe בין התהליכים בתור התקן.

<u>FIFOs</u> – pipe שדרכו יכולים כל התהליכים במכונה לגשת אליו. השימוש העיקרי בו הוא כאשר תהליכים רוצים לתקשר דרך ערוץ קבוע מראש מבלי שיהיה ביניהם קשרי משפחה.
לאחר היווצרו (קריאת מערכת ()mkfifo), ניתן לגשת ל-FIFO באמצעות פקודת ()open ולעבוד איתו כרגיל לקריאה וכתיבה (דרך אותו descriptor). תהליך שפותח את ה-FIFO לקריאה בלבד נחסם עד שתהליך נוסף יפתח את ה-FIFO לכתיבה, ולהיפך. הוא אינו מפונה בצורה אוטומטית לאחר שהמשתמש האחרון בו סוגר את הקובץ, ולכן יש לפנות אותו בצורה מפורשת באמצעות פקודות או קריאות מערכת למחיקת קבצים.

Signals

הודעה על אירוע שנשלח לתהליך בצורה אסינכרונית, שיכולה להישלח לתהליך בכל נקודה בזמן ללא תלות במצב התהליך.
ה-signals משמשים לתקשורת בין תהליכים ולהודעות ממערכת ההפעלה לתהליך על אירועים שונים.
כל סוג signal מוגדר לפי שם מתאים וערך מספרי מתאים ומיועד להודעה על סוג מסוים של אירועים.
יש 31 signals שמוגדרים ב-Linux, כאשר המשתמש יכול להגדיר בעצמו את 30 ו-31.
Signal שנשלח לתהליך כלשהו ולא טופל נקרא pending signal והוא נשמר ע"י מערכת ההפעלה עד לטיפול בו, ואחר כך הוא נמחק. יכול להיות לכל היותר pending signal אחד מכל סוג לתהליך נתון, ואם יתקבל signal נוסף מאותו ערך שעדיין לא טופל, המערכת תתעלם ממנו. בדיקה וטיפול ב-pending signal מבוצעת כל פעם שהתהליך עובר מ-Kernel Mode ל-User Mode במהלך הריצה שלו. תהליך ממתין יחזור למצב ריצה אם יש לו סיגנלים שממתינים לו.

<u>טיפול בסיגנלים</u> – תהליך יכול לטפל ב-signal בכמה אופנים שונים:

- **Terminate** – סיום התהליך בתגובה לאות.
- **Dump** – יצירת קובץ dump בשם "core" בספריית העבודה הנוכחית וסיום התהליך.
- **Ignore** – התעלמות מה-signal והמשך ביצוע התהליך כרגיל.
- **Stop** – עצירת התהליך במצב TASK_STOPPED.
- **Continue** – המשך ביצוע התהליך שהיה במצב TASK_STOPPED.
- **Signal handler** – הפעלת שגרת משתמש מיוחדת בתגובה ל-signal. השגרה מוגדרת בקוד התוכנית ומבוצעת ב-User Mode, בהקשר של התהליך שקיבל את ה-signal. הקשר הביצוע של התהליך נשמר לפני התחלת ביצוע השגרה והוא משוחזר אחרי סיומה, אבל לא מתרחשת החלפת הקשר. במהלך ביצוע השדרה נחסם זמנית טיפול ב-signals מהסוג שגרם לביצוע השגרה.

סיגנלים מהסוג SIGKILL (אילוץ סיום תוכנית) ו-SIGSTOP (עצירת תוכנית בשליטת debugger) לא ניתן לתפוס או לחסום.

תקשורת ומנגנוני תקשורת ב-Linux

רשת תקשורת היא אוסף חיבורים המאפשר תקשורת בין מחשבים. העברת נתונים ברשת תקשורת מבוצעת באמצעות פרוטוקולי רשת, כאשר הפרוטוקולים פועלים בשכבות שונות כשלכל שכבה תפקיד משלה. אוסף פרוטוקולי התקשורת הנפוץ ביותר הוא TCP/IP.

<u>מודל השכבות</u> – מודל סטנדרטי המאפשר לחלק את פרוטוקולי התקשורת לקבוצות או שכבות, כאשר לכל שכבה תפקיד משלה. מודל השכבות של TCP/IP מכיל 4 שכבות, כאשר כל שכבה מקבלת שירותים מהשכבה שמתחתיה, מספקת שירותים לשכבה שמעליה ומדברת עם השכבה המקבילה במחשב השני.

<u>שכבת הקו – Data Link</u> – תפקידה להעביר חבילה מתחנה נוכחית לתחנה שכנה באותה רשת, כאשר היא מטפלת בכל הפרטים הקשורים בחומרה ובממשק עם הרשת הפיסית. הפרוטוקול בשכבה זו שונה מרשת לרשת ותלוי בחומרה של הרשת הפיסית.

<u>שכבת Internet – פרוטוקול IP</u> – אחראית על ניתוב חבילות מתחנת המקור לתחנת היעד באמצעות פרוטוקול IP, שמאפשר ניתוב חבילות בין רשתות שונות. הדבר מתבצע באמצעות Gateways, שהם מחשבים מיוחדים שמחברים בין רשתות שונות ומאפשרים העברת חבילות בין הרשתות. הפרוטוקול לא מבטיח שחבילה תגיע אל יעדה, אלא רק מחליט איך לנתב אותה, והוא מנתב כל חבילה בנפרד, בלי קשר לחבילות האחרות. הפרוטוקול מבצע את הניתוב לפי כתובת IP שהיא מספר המזהה

בצורה חד ערכית כל ממשק תקשורת שמחובר לרשת, ושגודלה 32 ביט.
ההחלטה איך לנתב חבילה מתקבלת על סמך טבלאות ניתוב מיוחדות שקיימות בכל מחשב שיכול לשמש כנתב. נתב (router) הוא מחשב שדרכו עוברות חבילות תקשורת שלא מיועדות למחשב עצמו ובדרך כלל יש לו מספר ממשקי תקשורת. טבלאות הניתוב מאפשרות להחליט לאן לשלוח את החבילה לפי כתובת ה-IP שלה.

שכבת Transport – מאפשרת תקשורת ברמה של תהליכים. בשכבה זו שני פרוטוקולים עיקריים: UDP ו-TCP.
פרוטוקול TCP יוצר session בין שני תהליכים ומבטיח שהחבילות יגיעו ליעדן בסדר שבו נשלחו ושכל החבילות יגיעו ליעדן (אם החבילה הלכה לאיבוד היא משודרת מחדש וה-TCP מעכב את קבלת החבילה הבאה עד שהחבילה תגיע לבסוף).
כדי שיהיה ניתן לזהות חבילות ששייכות לאפליקציות מסוימות משתמשים ב-port, שהוא מושג וירטואלי המאפשר שימוש ברשת ע"י מספר אפליקציות במקביל. עבור כל אחד מהפרוטוקולים מוגדרת סדרה של פורטים, כאשר חלקם שמורים לאפליקציות מסוימות, ויישומי משתמש יכולים להשתמש רק במספרי פורטים הגדולים מ-1023. תהליך שמעוניין לאפשר לתהליכים מרוחקים להתחבר אליו מגדיר מספר פורט שעליו הוא מקשיב, כאשר רק תהליך אחד יכול להגדיר פורט חדש להקשיב עליו (אבל אב יכול לשתף פורט עם בניו). תהליך שרוצה להתחבר לתהליך שמקשיב, שולח חבילה שכתובת היעד שלה היא כתובת ה-IP של המחשב עליו רץ המקשיב, ומספר פורט היעד הוא מספר הפורט עליו התהליך מקשיב. שכבת ה-IP תעביר את החבילה למחשב היעד לפי כתובת היעד ואז שכבת ה-transport במחשב היעד תקבל את החבילה ומזהה לאיזה תהליך להעביר את החבילה לפי מספר הפורט.

שכבת האפליקציה – מכילה פרוטוקולים של אפליקציות (http, ftp) שמשתמשות בפרוטוקולי transport לתקשורת עם מחשבים מרוחקים.

מימוש של תקשורת ב-Linux

Socket – סוג של file descriptor שמתחבר לשכבת האפליקציה ולא לקבצים. הוא מאפשר לאפליקציה להבדיל בין sessions שונים (בקשות התחברות שונות), כאשר לכל session מתאים זוג socket בשרת ובלקוח.

יצירת ה-connection אינה סימטרית. תהליך אחד (השרת) יוצר socket, מחבר אותו לפורט עליו הוא מאזין ומחכה על ה-socket לבקשות תקשורת נכנסות. תהליך שני (הלקוח) יוצר socket ומנסה להתחבר אל השרת (כתובת ה-IP והפורט של השרת). מהרגע שה-connection הוקם, התקשורת תהיה סימטרית כאשר שני הצדדים יכולים לשלוח הודעות אחד לשני ע"י כתיבה ל-socket שלהם.

מודל שרת/לקוח (client/server) – מודל שרת/לקוח מתאר מצב שבו תהליך אחד (הלקוח) מבקש שירות מתהליך אחר (השרת). קיים מגוון רחב של שרתים, כאשר השרת יודע לטפל בדרך כלל בבקשות של מספר לקוחות בו זמנית.

פסיקות

פסיקות הן אירועים הגורמים למעבד להשעות את פעילותו הרגילה ולבצע פעילות מיוחדת.
השימושים האפשריים הם מימוש קריאות מערכת הפעלה, קבלת מידע וטיפול ברכיבי חומרה, טיפול בתקלות או אכיפת חלוקת זמן מעבד בין תהליכים.

סוגי פסיקות –

- **אסינכרוניות** – לא צפויות, נוצרות ע"י רכיבי חומרה שונים ללא תלות בפעילותו הנוכחית של המעבד.
- **סינכרוניות** – נוצרות בשל פעילותו של המעבד, למשל כתוצאה מתקלות שונות. נקראות גם חגירות (expeptions).
- **פסיקות תוכנה (או יזומות)** – פסיקות סינכרוניות שנוצרות ע"י הוראות מעבד מיוחדות ומשתמשים בהם למימוש קריאות מערכת או ל-debugging.

ביצוע הוראות מסוימות יכול להיכשל ולגרום לפסיקה סינכרונית. מעבדים מודרניים מאפשרים למערכת ההפעלה לתת להוראות הזדמנות שנייה להתבצע. הדבר שימושי למימוש זיכרון וירטואלי.

<u>גילוי וטיפול בפסיקה</u> – אם המעבד מגלה שהיו פסיקות לאחר ביצוע פעולה הוא שומר על המחסנית את כתובת החזרה (בדר"כ כתובת ההוראה הבאה, כאשר ב-Linux הערכים נשמרים על מחסנית הגרעין של התהליך ולא מחסנית המשתמש), מידע נוסף כמו רגיסטרים מיוחדים ומידע נוסף לגבי הפסיקה ומפעיל את שגרת הטיפול של הפסיקה. השגרה מוגדרת עבור התקן ופסיקה ולא רק עבור פסיקה בלבד, כאשר מספר התקנים יכולים לחלוק את אותה פסיקה וב-Linux ניתן להפעיל מספר שגרות בעקבות פסיקה אחת.
לכל פסיקה שגרת טיפול משלה שפועלת בהתאם לסוג הפסיקה. לכל פסיקה מספר שונה, ולמעבד יש גישה לטבלה של מצביעים לשגרות טיפול בפסיקה. מספר הכניסות בטבלה הוא כמספר וקטורי הפסיקות (במערכות IA32 יש 256 כניסות). הטבלה מאותחלת ע"י חומרת המחשב, ובטעינה, מערכת ההפעלה מעדכנת אותה ומחליפה את השגרות.
קוד הטיפול של הפסיקה מורץ בהקשר של התהליך הנוכחי כאשר לא מתבצעת החלפת תהליכים כדי לטפל בפסיקה. בדרך כלל, קוד הטיפול של הפסיקה לא מתייחס כלל לתהליך הנוכחי אלא למבני נתונים גלובליים של המערכת.

<u>בעיות בטיפול בפסיקות</u> – פסיקות א-סינכרוניות יכולות להגיע בכל זמן. מצד אחד, חשוב לטפל בהן בהקדם האפשרי כדי לעכב התקני קלט/פלט אלם זמן הטיפול עשוי להיות רב ולהתבצע על חשבון התהליך הנוכחי. בעיה נוספת היא שבזמן טיפול בפסיקה יכול להתרחש פסיקה נוספת.
פתרון חלקי הוא טיפול דו-שלבי בפסיקות א-סינכרוניות. החלק העליון (top half) מורץ כשגרת הטיפול בפסיקה באופן מהיר ככל האפשר ורושם בתור מיוחד את העובדה שיש לתת לחומרה הרלוונטית טיפול הולם. החלק התחתון (bottom half) מורץ בזמן מאוחר יותר ומבצע את העבודה השחורה של הטיפול בפסיקה, ופסיקות אחרות יכולות להתרחש תוך כדי פעולתו. המערכת שומרת תור של שגרות להרצה, כאשר שגרת הטיפול בפסיקה מוסיפה לתור את החלק התחתון המתאים.
בנקודות זמן מסוימות המערכת בודקת אם התור אינו ריק, ומריצה את כל השגרות המתאימות.
ניתן לשנות את שגרת הטיפול בפסיקה בזמן פעולת המערכת ע"י שינוי הטבלה. אם נרצה להוסיף לשגרה הנוכחית מבלי לבטל אותה, יש לשמור את כתובת השגרה הנוכחית לפני עדכון טבלת הפסיקות. שגרת הטיפול החדשה קוראת לשגרה הישנה, ע"י קריאה לשגרה המקורית ואחר כך מבצעים את השגרה החדשה, או קודם מבצעים את השגרה החדשה ורק אחר כך את המקורית (מאפשר לא לבצע את הפעולה המקורית במקרים מסוימים).

פסיקות ב-Linux

הטיפול בפסיקות ב-Linux הוא בתחום אחריותו של הגרעין. בתגובה על פסיקה מבוצע מסלול בקרה בגרעין, והגרעין חייב לסנכרן את הגישה למבני הנתונים שלו כדי להגן עליהם מפני מסלולי בקרה מקוננים או מקבילים. הפסיקה תטופל תמיד בהקשר הביצוע של התהליך הנוכחי, גם אם לו קשר לפסיקה שקרתה (וזמן הטיפול יחוסר מה-time slice שלו). המעבד בודק אם יש פסיקות ממתינות לטיפול בין ביצוע של כל שתי הוראות עוקבות בקוד, והטיפול בפסיקה מצריך מעבר ל-Kernel Mode כולל החלפת מחסניות לפי הצורך עבור התהליך הנוכחי.

Interrupts

נשלחות אל המעבד באופן א-סינכרוני ע"י התקני חומרה חיצוניים, וחלקן ניתנות לחסימה ע"י דגל מיוחד ברגיסטר הדגלים (IF). פסיקת חסומה לא תטופל עד שהחסימה תוסר (אם מדובר בפסיקת חומרה, החסימה מבוצעת אוטומטית). פסיקות מסוימות לא ניתנות לחסימות כמו פסיקות המדווחות על בעיית חומרה קריטיות.

הפסיקה מועברת אל המעבד באמצעות בקר פסיקות מתוכנת (APIC) נפרד לכל מעבד. כל התקן חומרה המבקש לשלוח פסיקה שולח אות IRQ לאחד מקווי הכניסה של בקר הכניסות אליו ההתקן מחובר. קיימים 16-24 קווי כניסות ממוספרים, כאשר אפשר לחבר כמה התקנים לאותו הקו (טיפול בפסיקת חומרה מחייב בדיקת כל ההתקנים שיכלו לגרום לה). כאשר בקר הפסיקות מבחין בבקשה מהתקן חומרה, הוא מעביר אות פסיקה למעבד אליו הוא מחובר, כאשר מספר הפסיקה הנוצרת עבור קו IRQ*n* ניתן לבחירה (בדרך כלל n+32).

Exceptions

פסיקות סינכרוניות (חריגות) שנוצרו ע"י המעבד כתוצאה מביצוע הוראה אחרונה בקוד ולא ע"י רכיבים חיצוניים, ואינן תלויות בדגל הפסיקות. קיימים 3 סוגים של חריגות:

1. **Faults** – מציינת תקלות הניתנות לתיקון בביצוע ההוראה האחרונה בקוד (כמו חלוקה ב-0 או גישה לא נכונה לזיכרון). כתובת החזרה בסיום הטיפול היא זו של ההוראה שגרמה לתקלה, כדי לבצע אותה מחדש.
2. **Traps** – נגרמות באופן מכוון ע"י ההוראה האחרונה בקוד, כדי להפעיל את קוד הטיפול בפסיקה. כתובת החזרה בסיום הטיפול היא זו שאחרי ההוראה שגרמה לפסיקה.
3. **Aborts** – מציינת תקלות חמורות בביצוע ההוראה האחרונה בקוד.

חריגות מתוכנתות או פסיקות תוכנה הן סוג של traps המשמשות לביצוע קריאות מערכת.

<u>(IDT) Interrupt Descriptor Table</u> – טבלה המקשרת בין וקטור פסיקה לשגרת הטיפול בה. יש בה 256 רשומות כשכל רשומה בגודל של 8 בתים. כל רשומה בטבלה מכילה את סוג הרשומה, כתובת שגרת הטיפול, וערך DPL שמייצג מהו ערך ה-CPL המקסימלי להרצת שגרת הטיפול.

קיימות 3 סוגי רשומות ב-IDT:

1. **Interrupt Gate** – לפסיקות חומרה. דגל ה-IF מכובה אוטומטית בגישה לטיפול דרך רשומה זו.
2. **Trap Gate** – לחריגות, אין שינוי ב-IF.
3. **Task Gate** – מגדיר תהליך שיוזמן לטיפול בפסיקה. אינו בשימוש ב-Linux, כי הגרעין מטפל בפסיקות.

<u>אתחול ה-IDT ב-Linux</u> – במקור, הטבלה מאותחלת ע"י ה-BIOS,אולם Linux מחליפה את כל הטבלה בזמן הטעינה ע"י הפעלת פונקציה (detup_idt שמאתחלת את כל הרשומות בטבלה לערכי ברירת מחדל (Interrupt Gate, כתובת השגרה (ignore_int שמדפיסה הודעת שגיאה וקובעת DPL=0).
לאחר האתחול, מתבצע מעבר נוסף על ה-IDT על מנת לעדכן את הכניסות המתאימות לטיפול בחריגות ובפסיקות מהחומרה המותקנת.

<u>טיפול בחריגות</u> – מרבית שגרות הטיפול הפנימי בחריגות שולחות signal לתהליך הנוכחי כדי שיטפל בתקלה שנגרמה, כאשר סוג ה-signal נקבע לפי סוג החריגה המטופלת, וזה יתגלה לפני החזרה ל-User Mode, ואז יתבצע הטיפול ב-signal.

טיפול בפסיקות חומרה ב-Linux

הטיפול בכל פסיקות החומרה עובר דרך מנגנון מרכזי אחד המתחיל ב-common_interrupt, כאשר האבחנה בין הפסיקות היא באמצעות הערך הנשמר במחסנית מיד בתחילת שגרת הפסיקה.
הפונקציה **do_IRQ()** – מבצעת את הטיפול בפסיקה ברמת תפעול ה-APIC וקוראת לפונקציה מיוחדת להפעלת ה-ISRs. ראשית, היא שולפת את סוג הפסיקה המטופלת, שולחת אישור סלקטיבי ל-APIC, מפעילה את שגרות טיפול הפסיקה, מאפשרת סוג הפסיקה המטופלת ב-APIC ומבצעת משימות ממתינות.
טיפול ב-APIC – כאשר מתרחשת פסיקת חומרה, ה-APIC מודיע עליה למעבד. ה-APIC ננעל ולא מודיע על פסיקות נוספות עד שיקבל אישור מהמעבד. מקובל לשלוח לו בהקדם אישור להודיע על פסיקות חדשות פרט לכאלה מהסוג המטופל כרגע, וכך יהיה נתן לטפל במסלולי בקרה אחרים בפסיקות נוספות, אולם למנוע כניסה כפולה לקוד שמטפל בפסיקה, דבר שיכול לגרום לשינויים לא רצויים במשתנים הגלובליים. בסיום הטיפול בפסיקה, מאפשרים ל-APIC להודיע גם על פסיקה מהסוג שטופל.
הפעלת ה-ISRs – כל ISR מיועד בדרך כלל לטיפול בהתקן מסוים בתגובה לפסיקת חומרה מסוימת. בזמן הפעלת שגרת הטיפול בפסיקות, מפעילים את כל ה-ISRs הרשומים עבור הפסיקה הנוכחית בזה אחר זה, כדי לבדוק מי ההתקן שגרם לפסיקה.

הטיפול בפסיקות חומרה עשוי לדרוש זמן רב, כאשר ביצוע שגרת טיפול במלואה בפסיקות חסומות עלול לגרום לעיכוב הטיפול בפסיקות אחרות, דבר שיפגע בביצועים ויגרום לאובדן פסיקות או נתונים. מנסים לצמצם למינימום את הפעילות הנעשית בפסיקות חסומות, ולצורך כך קיימות דרגות דחיפות בטיפול בפסיקות חומרה:

1. **Critical** – פעולות שהכרחי לבצען בפסיקות חסומות, והן מתבצעות במהירות.
2. **Noncritical** – פעולות שיש לבצען מיד במסגרת הפסיקה, אך אין צורך בחסימת הפסיקות (יבוצע באמצעות ה-ISR).
3. **Noncritical deferred** – משימות שיש צורך לבצען בעקבות הפסיקה, אך לא באופן מיידי, אלא כשיסתיים הטיפול המיידי בפסיקה. משימות אלו מוכנסות למאגר של משימות ממתינות לביצוע בעתיד.

סוגי משימות ממתינות –

1. **Softirq** – מנגנון בסיסי, מממש את הסוגים האחרים. פונקציה שמתבצעת ב-softirq יכולה להיות מופעלת במקביל משני מעבדים שונים (חייבת לפעול במקביליות ולא לגשת למשתנים גלובליים).
2. **Tasklet** – המנגנון המומלץ, לשימוש רגיל. הקוד מוגן מפני הפעלה במקביל ממעבדים שונים.
3. **Bottom Halves** – tasklets בעדיפות גבוהה. כל המשימות מהסוג הזה מהוות קטע קריטי יחיד, כלומר שני bottom halves (אפילו עם קוד שונה) לא מתבצעים במקביל ממעבדים שונים.

בכל מעבד יחיד, המשימות הממתינות מתבצעות באופן סדרתי, ואף משימה לא מתחילה להתבצע לפני סיום הקודמת. בנקודות זמן שונות בודקים אם יש משימות ממתינות לביצוע.

מחסנית הגרעין – סדר שמירת האיברים במחסנית אחיד וקבוע בכל אחת מצורות הטיפול, כאשר כל האיברים נשמרים כשדות של 32 ביט, כולל תכולת ה-segment registers. באופן זה ניתן לקפל את מחסנית הגרעין באותה צורה בעת חזרה מטיפול בכל סוג פסיקה שהוא.
החלק הגבוה של המחסנית (cs:eip ,eflags ,ss:esp) נשמר באופן אוטומטי ע"י המעבד, ואילו שאר האיברים מוכנסים למחסנית מיד בתחילת שגרת הפסיקה.

החלפת הקשר בזמן טיפול בפסיקות – בזמן טיפול בפסיקה א-סינכרונית לא ניתן לבצע החלפת הקשר ב-Linux, אבל מסמנים כי נדרשת החלפת הקשר באמצעות הדלקת הדגל need_resched במתאר התהליך. בסיום הטיפול בפסיקה לפני החזרה ל-User Mode, בודקים אם צריך להחליף הקשר.

סיום טיפול בפסיקות – כשטיפול בפסיקה מסתיים בודקים אם הפסיקה הינה מקוננת, האם צריך להחליף הקשר ואם יש signals ממתינים שדורשים טיפול. רק לאחר מכן משחזרים את ערכי הרגיסטרים מלפני הפסיקה וחוזרים לתהליך שבזמן פעולתו התרחשה הפסיקה.

מנהלי התקנים ב-Linux

מנהל התקן – שכבת תוכנה החוצצת בין החומרה לבין האפליקציה. מנהל ההתקן מחביא את הפרטים הנוגעים לפעולת החומרה וממפה את הפונקציות הסטנדרטיות לפעולות המבוצעות על התקן החומרה, בעוד המשתמש מבצע פעולות באמצעות אוסף פונקציות סטנדרטי שאינו תלוי בהתקן מסוים. הגישה היא מודולרית, כאשר ניתן לבנות מנהל התקן בנפרד משאר הגרעין ואז לחבר אותו בזמן ריצה, בשעת הצורך.

מודול – Linux יכולה להרחיב בזמן ריצה את אוסף הפעולות שמספק הגרעין, וקטע קוד שניתן להוסיף לגרעין בזמן ריצה נקרא מודול, והוא למעשה ספריה משותפת הנטענת לגרעין (מקושרת אליו בזמן ריצה). רק משתמשים מורשים יכולים לטעון מודולים.

קיימים סוגים שונים של מודולים לפי התפקודיות שהם מספקים:

- **התקן תווים** – התקן שניגשים אליו כאל רצף של בתים (כמו קובץ). בדרך כלל ניתן לגשת להתקן תווים רק באופן סדרתי, ורק חלק מההתקנים מאפשרים לגשת בצורה של seek (חיפוש לאורך הזיכרון). ניתן לגשת אליו דרך מערכת הקבצים.
- **התקן בלוקים** – התקן שניתן לתקשר איתו רק בכפולות של בלוק (למשל Kbyte). בפועל Linux מאפשרת לקרוא מהתקני בלוקים גם בבתים בודדים. הם משמשים, בין השאר, לייצוג מערכות קבצים וניתן לגשת אליהם דרך מערכת הקבצים.
- **ממשקי רשת** – התקנים המסוגלים להחליף מידע עם מחשבים אחרים, ובאחריותם לשלוח ולקבל חבילות מידע. הם אינם מיוצגים במערכת הקבצים.
- **ועוד**... (USB וכו').

אבטחה – מודול רץ ב-kernel mode ולכן יש גישה למבני הנתונים של הגרעין. חשוב להימנע מחורי אבטחה במודול כדי שלא נדרוס מידע חשוב בגרעין, להקפיד על אתחול משתנים, לבדוק את קלט המשתמש, ובמידת הצורך להגביל את השימוש במודול למשתמשים מורשים.

העברת פרמטרים למודול – בעת טעינת המודול, ניתן להעביר אליו פרמטרים הנוגעים לקונפיגורציות של המודול. בקוד המודול מגדירים את המשתנים שיקבלו את הפרמטרים באמצעות המאקרו MODULE_PARM, שצריך להופיע מחוץ לפונקציה (בדרך כלל בתחילת המודול).

התקני תווים – ניגשים אליהם דרך שמות המופיעים במערכת הקבצים. התקן תווים מאופיין ע"י שני מספרים:

- **מספר ראשי** – מזהה את מנהל ההתקן המקושר להתקן, ולמעשה קובע את הדרייבר של ההתקן.
- **מספר משני** – מספר שמשמש את מנהל ההתקן כדי להבחין בין התקנים שונים המחוברים אליו.

הוספת מנהל התקן חדש – כדי ליצור מנהל התקן חדש, יש להקצות לו מספר ראשי באמצעות הפונקציה register_chrdev(), והיא מבוצעת במסגרת אתחול מנהל ההתקן. גרסה 2.4 של Linux תומכת ב-256 מספרים ראשיים, כאשר חלק מהם מוקצים באופן סטטי להתקנים נפוצים. הצקאה סטטית עבור כל התקן יכולה להיות בעייתית כי יכולה להיווצר התנגשות בין מספרים ראשיים של התקנים שונים, ולכן ניתן לבצע הקצאה דינמית של מספר ראשי ע"י העברת הערך 0 כפרמטר לפונקציה. כשמסירים מנהל התקן מהמערכת, יש לשחרר את המספר הראשי שהוקצה לו.

יצירת התקן חדש – הקריאה ל-register_chrdev() רק מקשרת בין מנהל ההתקן לבין מספר ראשי, וכדי לעבוד עם ההתקן, יש ליצור קובץ חדש באמצעות הפקודה mknod.
קבצים מיוצגים ע"י מבנה נתונים בשם inode, ובפרט גם להתקן תווים יש inode המייצג אותו. בכל פעם שמבצעים פעולה על התקן, מנהל ההתקן מקבל כפרמטר את ה-ionode המתאים.
מערכת ההפעלה מדגירה אוסף פעולות שניתן לבצע על קבצים, ובפרט על התקן תווים. מבנה הנתונים file_operations הוא מערך של מצביעי הפונקציות המממשות את אותן הפעולות.
הפונקציות הן:

- **Open** – כאשר פותחים התקן, מערכת ההפעלה מאתחלת מבנה נתונים מסוג struct file שמייצג קובץ/התקן פתוח, כאשר כל הקריאות שהמשתמש מבצע על file descriptor מסוים מועברים למנהל ההתקן עם אותו struct file.מצביע למבנה הנתונים הנ"ל נקרא filp או file. במידה והפונקציה open אותחלה עבור מנהל ההתקן, מערכת ההפעלה קוראת לה ומעבירה את מבנה הנתונים כפרמטר. הפונקציה אחראית להגדלת מונה השימוש של מנהל ההתקן, בדיקה שאין בעיות חומרה והמנהל תקין, מאתחלת את ההתקן אם זו הפעם הראשונה שפותחים אותו, מזהה את המספר המשני ומאתחלת מבני נתונים ב-filp->private_data.
- **Release** – אחראית להקטנת מונה השימוש של מנהל ההתקן ולשחרור מבני נתונים שהוגדרו ב-filp->private_data. פעולת סגירה מתבצעת גם אם האפליקציה לא סגרה קבצים פתוחים באופן מפורש.
- **Flush** – לניקוי הזיכרון להתקן.
- **Read ו-Write** – לקריאה וכתיבה מההתקן. הן מוודאות שהגישה מתבצעת לאזורים חוקיים בזיכרון, אחרת לא מתבצעת כל העתקה, ובכל מקרה מוחזר מספר הבתים שנותרו להעתקה. ניתן להקצות ולשחרר זיכרון בגרעין באמצעות kmalloc ו-kfree.
- **Llseek** – בודקת האם ניתן להזיז את סמן הקובץ.
- **Ioctl** – מטרתה לאפשר פקודות בקרה ייחודיות להתקן, ויש להשתמש בה רק אם לא ניתן לספק מענה הולם במסגרת הפונקציות הקיימות. עדיף שהתקנים שונים יגדירו קבועים עבור הפרמטר שמציין את המספר הסידורי של פקודה, על מנת למנוע תקלות שיכולות לקרות בגלל שהתקנים שונים עושים שימוש באותם מספרים.

מערכת הקבצים

קבצים מאפשרים שמירת מידע לטווח בינוני ואורך, למרות הפעלות חוזרות של תוכניות, אתחולים ונפילות. מטרות מערכת הקבצים היא להפשיט מתכונות ספציפיות של ההתקן הפיזי, זמן ביצוע סביר, ארגון נוח של הנתונים, הפרדה בין משתמשים שונים והגנה.

קובץ – מידע עם תכונות המנוהלות ע"י המערכת. הוא מכיל מספק בתים/שורות/רשומות בגודל קבוע או משתנה, והוא יכול להיות מסוג מסוים וע"י כך להיות מזוהה ע"י מערכת ההפעלה. מערכת הקבצים תגדיר לו שם, גודל (בדרך כלל בבתים), מיקום, מידע על בעלות והגנה ותוויות זמן.

פעולות על קבצים

- **Create** – יצירת קובץ, הקצאת מקום בשבילו ושמירת התכונות שלו.

- **Delete** – שחרור המקום שהוקצה לקובץ יחד עם התכונות שלו.
- **Read** – נותנת שם קובץ, כמות מידע לקריאה וחוצץ זיכרון שבו יאוחסן המידע הנקרא. הפעולה מתחזקת מצביע מיקום לקובץ.
- **Write** – נותנת שם קובץ ומידע לכתיבה אליו, ומתחזקת מצביע מיקום לקובץ.
- **Seek** – הזזת מיקום המצביע.
- **Open, Close** – מאחזרות מידע על הקובץ ומאתחלות את מצביע המיקום לקובץ (יצירת מבנה נתונים לקובץ), וע"י כך משפרות את ביצועי המערכת.
- **Append, Rename, Copy**.
- **נעילה**
- **עדכון תכונות הקובץ.**

כאשר כותבים לקבוץ, לא כותבים בפועל אל הדיסק, אלא ל-buffer (חוצץ) כדי למנוע יותר מדי גישות לדיסק (פעולה איטית). ביצוע close בודק שהעתקנו הכל לדיסק ומשחרר את הזיכרון.

אופני גישה לקובץ – ניתן לגשת לקבצים בגישה סדרתית (לפי סדר, בדרך כלל מהירה ולא צריך לציין מהיכן לקרוא וע"י כך מאפשרים למערכת להביא נתונים מראש) או בגישה ישירה או אקראית (לפי מיקום או לפי מפתח).

מדריכים – מבנה נתונים מופשט ששומר את התכונות של כל קובץ הנמצא בו. מדריך תומך במציאת קובץ (לפי שם), יצירה או מחיקה של כניסה, קבלת רשימת הקבצים בתוך המדריך, החלפת שם של קובץ ועוד. קיימים 2 סוגים עיקריים של מבני מדריכים:

- **עץ מכוון** – עץ בעל מספר רמות שרירותי. קובץ מזוהה ע"י מסלול מהשורש (מוחלט) או מהמדריך הנוכחי (יחסי). בשימוש במערכות הפעלה מבוססות MS-DOS.
- **גרף אציקלי** – מאפשרים למספר כניסות במדריכים להצביע לאותם העצמים. בשימוש במערכות מבוססות UNIX.

סוגי קישורים –

- **קישור רך** – כניסה שמכילה את המסלול לקובץ. קישור זה אינו מונע מחיקה מלאה של קובץ, אלא משאיר מצביע אליו תלוי באוויר.
- **קישור חזק** – לא ניתן להבדיל מהכניסה המקורית. הוא מחייב מחיקת כל הכניסות במקרה של מחיקה קובץ.

מבנה מדריכים ב-UNIX – לכל קובץ ניתן להגיע דרך מסלולים שונים, כאשר שם (מסלול מלא) אינו תכונה של הקובץ. מחזיקים use-counter שמאפשר לדעת מתי למחוק קובץ, כאשר אין אליו קישורים. לתהליך מבנה נתונים של קבצים פתוחים, שכולל מידע על מיקום נוכחי בקובץ שאינו משותף עם פתיחות אחרות של אותו קובץ, כולל שם התהליך.

הגנה – כדי למנוע ממשתמש לבצע פעולות ספציפיות על הקובץ, קבעו לכל קובץ רשימות גישה, כאשר הן מגדירות לכל קובץ למי מותר לבצע איזה פעולה. כדי למנוע מהן להפוך לארוכות מדי וקשות לתחזוקה, מקבצים משתמשים למחלקות שונות ומגבילים את מספר הפעולות.
קיימים מספר סוגים של דרכים לנעול קבצים:

- **Advisory locking** – תהליך צריך לנעול קובץ בצורה מפורשת, כאשר המשתמש אחראי לנעילת הקובץ ואחראי לפתוח אותו. ניתן לנעול קובץ שלם או חלקים ממנו.
- **Mandatory locking** – אם הקובץ נעול, כל תהליך שמנסה לקרוא מהקובץ (באמצעות open, read או write) יינעל, גם אם לא ביצע נעילה מפורשת.

- **Leases** – אם תהליך שנעל את הקובץ לא משחרר אותו ולא מאריך את ה-lease תוך פרק זמן נתון, המנעול משתחרר.

היררכיית האחסון – רגיסטרים, מטמון וזיכרון ראשי (פיזי) הם מהירים אולם הם קטנים ויקרים. דיסק קשיח ואמצעי גיבוי כגון CD, DVD הם גדולים וזולים, אולם הם איטיים, והם תחת הניהול של מערכת ההפעלה. הם נקראים זיכרונות משניים, והם אינם מאפשרים ביצוע ישיר של פקודות או גישה לבתים.

מימוש מערכת הקבצים

מבנה מערכת קבצים טיפוסית

- **Logical file system** – קוד כללי, בלתי תלוי במערכת קבצים ספציפית. הקוד מקבל מסלול ומחזיר את קובץ היעד, ובנוסף יש ניהול מידע עבור קבצים פתוחים כמו מיקום בקובץ. יש פעולות על מדריכים יחד עם הגנה ובטיחות הקבצים.
- **Virtual file system interface** – מנשק אחיד לכל מערכות הקבצים הספציפיות.
- **Physical file system** – מימוש מנשק ה-VFS עבור מערכת קבצים ספציפית ומתכנן איך לפזר את הבלוקים.
- **Device drivers** – קוד שיודע איך לפנות להתקן ספציפי. מתחיל את הפעולה הפיזית ומטפל בסיומה, מתזמן את הגישות על מנת לשפר ביצועים.

המדדים להערכת מימוש מערכת הקבצים הם:

- מהירות גישה סדרתית.
- מהירות גישה אקראית (ישירה).
- שברור פנימי וחיצוני.
- יכולת להגדיל קובץ.
- התאוששות משיבושי מידע.

מיפוי קבצים

הקצאה רציפה – בבלוקים באורך קבוע (כפולה של גודל סקטור על הדיסק, נע בין 512B-4KB) כאשר המשתמש מצהיר על גודל הקובץ עם יצירתו. מחפשים בלוקים רצופים שיכולים להכיל את הקובץ והכניסה במדריך מצביעה לבלוק הראשון בקובץ.
היתרון הגדול הוא גישה מהירה (סדרתית וישירה), אולם יכול להתרחש שברור פנימי וחיצוני וקשה להגדיל קובץ.

הקצאה משורשת – כל בלוק מצביע לבלוק הבא, הכניסה במדריך מצביעה לבלוק הראשון בקובץ. קל להגדיל בשיטה זו את הקבצים ויש מעט שברור חיצוני. מצד שני, הגישה היא איטית (בעיקר לגישה ישירה) אולם השימוש בבלוקים גדולים מקטין את הבעיה, אבל הוא מגדיל את השברור הפנימי. שיבוש מצביע בבלוק יכול לגרום לאיבוד חלקים של קובץ.

(FAT) File Allocation Table – החזקת שרשרת המצביעים בנפרד. בעצם, משתמשים בטבלה שמתארת את התוכן של הדיסק כולו, והמצביע הקובץ (במדריך) מראה על הכניסה הראשונה).
השיטה ממומשת במערכת מבוססות MS-DOS.
שיטה זו מוגבלת כי הטבלה נמצאת במקום נוח בדיסק / זיכרון ראשי ומכילה כניסות (אינדקס של 16 ביטים, או 2^{32} בגרסת FAT-32. כאשר הדיסקים גדלים, גודל החתיכות גדל, דבר שמגדיל את השברור הפנימי. כל קובץ דורש לפחות חתיכה אחת, מה שמגביל את כמות הקבצים (4G ב--FAT 32). בנוסף, הטבלה מהווה משאב קריטי, ואיבוד שלה או פגיעה בה היא קטסטרופלית (לכן היא מוחזקת בשני עותקים).

אינדקס – המשתמש מצהיר על גודל קובץ מקסימלי, המצביעים לבלוקים מרוכזים בשטח רצוף. הגישה משפר במקצת את זמן הגישה הישירה לעומת הקצאה משורשרת, אולם צריך להצהיר מראש על גודל קובץ. במערכות מבוססות על אינדקס מרובה רמות, משתמשים במבנה נתונים הנקרא **inode** שמחזיק את כל המידע על הקובץ (שם, משתמשים, הרשאות, תוויות זמן, מספר הלינקים אל הקובץ ומצביעים ממנו). ה-inodes (ממומשים ב-Unix 4.1) אינם מדריכים, כאשר מדריכים הם קבצים רגילים שממפים שמות קבצים ל-inodes. לאחר שימוש ממושך במערכת הקבצים בלוקים שונים של אותו קובץ נמצאים רחוק זה מזה, ואילו inodes ובלוקי אינדקס נמצאים רחוק מבלוקים של מידע.

ניהול בלוקים פנויים

Bitmap – מערך ובו סיבית לכל בלוק (אם הסיבית דולקת, הבלוק פנוי). המערך מאוחסן במקום קבוע בדיסק, עם עותק בזיכרון הראשי, ליעילות. קל לזהות באמצעותו רצף של בלוקים פנויים.

רשימה מקושרת – מציאת בלוק פנוי בודד היא מהירה, ולכן נוכל להקצות מספר בלוקים לאותו קובץ כאשר הם מרוחקים זה מזה. מציאת הבלוקים מחייבת תזוזות של הדיסק, ומבנה הנתונים יכול להיהרס אם בלוק באמצע הרשימה השתבש. ב-FAT, הבלוקים הפנויים מנוהלים כקובץ אחד.

קיבוץ – שימוש ברשימה מקושרת של אלמנטים: כל אלמנט מכיל טבלה של מצביעים לבלוקים פנויים רצופים ומצביע לאלמנט הבא (grouping) וכל אלמנט מכיל מצביע לבלוק הפנוי הראשון מקבוצת בלוקים פנויים רצופים, מספר הבלוקים בקבוצה ומצביע לאלמנט הבא (counting). ניתן למצוא בצורה יעילה מספק גדול של בלוקים פנויים ורציפים.

אמינות

המידע בדיסק מתחלק ל-user data (נתוני המשתמש בקבצים) ו-metadata (מידע על ארגון הקבצים). איבוד או שיבוש של metadata יכול לגרום לאיבוד user data רב
במערכות Unix משתמשים במדיניות של write-back – הדיסק מתעדכן באופן א-סינכרוני, אולי לא לפי הסדר, נתוני המשתמש נכתבים באופן מחזורי לדיסק.
אמינות ה-metadata – שימוש במדיניות write-through, שלפיה נתונים ועדכונים נכתבים ישירות לדיסק. נתונים מסוימים נשמרים במספר עותקים (מס' מקומות בדיסק), כדי שאם יהיה מקום בדיסק שייהרס, נוכל לשחזר את הנתונים הקריטיים ממקום אחר שלא נפגע. כאשר מערכת הקבצים עולה אחרי נפילה, מתקנים את מבני הנתונים.

רישום (logging) – שיטה יעילה לתחזוקת ה-metadata. רושמים ב-log סידרתי את העדכונים לפני שהם נכתבים לדיסק, ולאחר מכן כותבים לדיסק את הבלוקים שהשתנו (לא חובה לפי הסדר, ואפשר לקבץ מספר שינויים ולכתוב אותם בכתיבה אחת). ניתן למחוק מה-log עדכונים שכבר נכתבו לדיסק. לאחר נפילה, עוברים על כל הכניסות ב-log ומבצעים ביצוע נוסף של פקודה שהתבצעה במלואה או חלקית, וזה נותן תוצאה שקולה לביצוע הפעולה המקורית. אם הכניסה האחרונה ב-log אינה שלמה, מתעלמים ממנה.
יתרונות השיטה הן כתיבה לדיסק באופן א-סינכרוני והתאוששת יעילה (כתלות במספר השינויים שלא נכתבו לדיסק ולא בגודל מערכת הקבצים). החיסרון הוא שהשיטה דורשת כתיבות נוספות.

מערכות קבצים מבוזרות

המטרה היא לאפשר למספר תהליכים שרצים במכונות שונות גישה ושיתוף קבצים, תוך כדי שקיפות לאפליקציה (אפליקציה שנכתבה לטיפול בקבצים מקומיים לא צריכה להשתנות), שקיפות מקום (שם הקובץ לא מכיל מידע על מיקומו) ואי תלות במקום (שינוי מיקום הקובץ לא נראה למשתמש).

תהליך רץ במחשב לקוח (client) ומבקש לגשת לקובץ הנמצא במחשב שרת (server). מחשב הלקוח ומחשב השרת מחוברים באמצעי תקשורת כלשהו, ובעקבות קריאות מערכת של תהליכים מהלקוח בעבודה עם קבצים מהשרת, הלקוח שולח לשרת בקשות והשרת מחזיר תשובות באמצעות תהליך

מיוחד שמאזין לבקשות, מבצע אותן ומחזיר תשובות. סוגי ההודעות בין הלקוח והשרת מוגדרים בפרוטוקול תקשורת.
השרת יכול לשמור מצב עבור כל לקוח בין בקשה לבקשה (איזה קבצים נפתחו, מיקום בקובץ, מידע על הנתונים במטמון של הלקוח, מנעולים וכו') כאשר בקשה מתבצעת בהקשר מסוים ואין צורך להעביר את הכל המידע הדרוש לביצוע הפקודה. קיים פרוטוקול בלי מצב שבו השרת לא שומר מידע ובקשה מכילה את כל המידע הנחוץ לטיפול בה. קל יותר לממש את הפרוטוקול הזה וקל יותר להתאושש מנפילות, אולם לא ניתן לבצע שיפורים ולחסוך בתקשורת וקשה לממש נעילות של קבצים.

Network File System (NFS) – פרוטוקול בין לקוח ושרת, שהוא בעיקרון ללא מצב. פעולות Remote Procedure Call (RPC) שמתבצעות עליו הן read ו-write על קובץ, גישה לתכונות של קובץ, חיפוש בתוך מדריך, פעולות על מדריכים כמו חיפוש, הוספה או מחיקה של כניסה וכד' (אין בפרוטוקול פעולות של open ו-close). הלקוח מרכיב (mounts) תת עץ של השרת במדריך שלו. הפרוטוקול שקוף לאפליקציה היות וניתן לגשת לקבצים בשרת כמו לקובץ מקומי, ויש שקיפות מקום היות והמשתמש לא מבחין מהו מיקום הקובץ. יש תלות במיקום כי הזזת קובץ מחייבת שינוי ה-mount. NFS מבוסס על שיגור הפונקציה לביצוע אל השרת, אך כדי לקבל ביצועים, משתמשים במטמון (cache) אצל הלקוח שאליו קוראים חלקים מהקובץ שאיתו עובדים. עדכונים לקובץ נאגרים במטמון ונשלחים לשרת מדי פרק זמן, אולם כתיבות לקובץ אינן משפיעות מיד אצל לקוחות אחרים. כמעט כל קריאת מערכת (לגישה לקובץ) מתורגמת ישירות לפעולת RPC.

מערכת קבצים ב-Linux

אנחנו מעוניינים שמערכת ההפעלה תוכל לחבר מערכות קבצים מסוגים שונים ותדע לעבוד איתם בצורה אחידה, כאשר הרעיון הוא להחזיק בגרעין מידע על מספר רב של מערכות קבצים שיאפשר תמיכה בכל פעולות המוגדרות ע"י כל אחת ממערכות הקבצים הנתמכות, ולהגדיר ממשק אחיד לכל המערכות, שישתמש בפעולות הספציפיות המוגדרות עבור כל מערכת ומערכת.

Virtual File System (VFS) – מערכת קבצים מדומה היא שכבת תוכנה בגרעין המספקת ממשק אחיד לכל מערכות הקבצים הספציפיות. היא מאפשרת להרכיב מערכות קבצים מסוגים שונים ומספקת ממשק אחיד לגישה לקבצים הנמצאים במערכות הקבצים השונות, ומתרגמת את קריאות המערכת הכלליות לקריאות ספציפיות המתאימות לאותה מערכת קבצים בה נמצא הקובץ.

מודל משותף לניהול קבצים (Common File Model) – VFS מגדירה מודל משותף לניהול קבצים עבור כל סוגי מערכות הקבצים הנתמכות ע"י Linux. המודל תואם את המודל המקובל בכל מערכות Unix כאשר הוא מצמצם את התקורה בניהול מערכת הקבצים הרגילה של Linux. כל מערכות הקבצים האחרות צריכות לתרגם את המבנה שלהם למודל המוגדר ע"י VFS. לפי המודל של VFS, מדריך הוא קובץ המכיל רשימה של קבצים ומדריכים אחרים, בעוד מערכת קבצים מסוג FAT מנהלת עץ מדריכים מיוחד והמדריכים הם אינם קבצים. כדי להתאים למודל המשותף, Linux מייצר קבצי מדריכים מהמבנים המוגדרים במערכת קבצים מסוג FAT.
VFS משתמש במספר סוגי אובייקטים לניהול העבודה עם מערכות הקבצים, מדריכים וקבצים:

- **Superblock object** – מכיל מידע כללי על מערכת קבצים מסוימת. הוא מכיל מידע על ההתקן שעליו מאוחסנת מערכת הקבצים, סוג מערכת הקבצים וכו', והוא כולל מבנה המכיל פונקציות טיפול עבור מערכת הקבצים הנתונה, שממממשות את הממשק המשותף של ה-VFS עבור מערכת הקבצים הנוכחית, ועבור כל מערכת קבצים מוגדר אוסף זהה של פונקציות טיפול, אבל רק חלק ממומש בהתאם לסוג מערכת הקבצים.
- **Inode object** – מכיל מידע על קובץ מסוים, כאשר מספר ה-inode מזהה בצורה חד משמעית את הקובץ במערכת הקבצים. הוא מאותחל עם המידע הנמצא ב-Inode התואם על הדיסק, ואם ערך השדות בו משתנה, צריך לעדכן גם את ה-inode התואם על הדיסק.

המידע בו כולל תכונות שונות של הקובץ, כמו הרשאות, בעלות, גודל, חותמת זמן ומצביעים לפונקציות המגדירות כיצד לטפל בקובץ.

- **File object** – מכיל מידע על קובץ שנפתח ע"י תהליך כלשהו. הוא מכיל את המיקום הנוכחי בקובץ ומצביע ל-inode המתאים ועל מבנה המצביע לפונקציות הטיפול בקובץ שנפתח. הוא מוגדר רק על קובץ פתוח ואין לו מבנה תואם על הדיסק.
- **Dentry objects** – מכיל מידע לגבי כניסה במדריך (שיכולה להיות מדריך או קובץ). הוא קיים רק בזיכרון הראשי ואין מבנה תואם על הדיסק. הוא מכיל מצביע ל-inode של הקובץ או המדריך אליו הכניסה מתייחסת ואם ה-dentry הוא של כניסה של תת מדריך, הוא מצביע לרשימה של הבנים של אותו תת מדריך. בהרבה מקרים תהליכים ניגשים לאותו קובץ בפרק זמן קצר. קריאת מידע המתאר מדריך מסוים מהדיסק ובניית dentry מתאים הן פעולות יקרות המצריכות זמן רק, ולכן שימוש במטמון של dentries מאפשר לצמצם את הזמן הדרוש ע"י שמירה של dentries בזיכרון.

<u>הרכבה של מערכת קבצים – Mounting</u> – ניתן להוסיף מערכות קבצים נוספות לתוך ה-VFS באמצעות פעולה של הרכבה. הרכבת מערכת הקבצים החדשה נעשית לתוך מדריך קיים במערכת הקבצים הקיימת (mount point). לכל מערכת קבצים פיזית מוגדר מדריך שורש, וכל מערכת קבצים שמורכבת לתוך השורש של ה-VFS נקראת מערכת הקבצים הראשית (root filesystem). ההרכבה נעשית באמצעות קריאת המערכת ()mount, ולאחר שמרכיבים למדריך מסוים מערכת קבצים חדשה, לא ניתן לגשת לקבצים ולמדריכים שהיו תחת מדריך זה לפני ביצוע ההרכבה. לאחר ההרכבה ניתן לגשת לקבצים במערכת הקבצים שהוספה באמצעות שרשור המסלול מהשורש לנקודת ההרכבה, יחד עם המסלול מהשורש של המערכת המורכבת אל הקובץ המבוקש. ב-Linux ניתן להרכיב למערכת הקבצים הכללית את אותה מערכת קבצים פיזית מספר פעמים בנקודת הרכבה שונות.
ניתוק מערכת קבצים מה-VFS נעשה ע"י קריאת המערכת ()unmount, שתצליח רק במקרה ולנותן ההוראה יש את ההרשאות המתאימות. אם למערכת הקבצים שמנתקים מורכבות מערכות קבצים אחרות, גם הן יתנתקו. לאחר הניתוק יהיה ניתן שוב לגשת אל הקבצים שהיו תחת נקודת ההרכבה לפני ההרכבה.

<u>איתור מסלול (pathname)</u> – כדי להגיע ל- inode המתאים של הקובץ ניתן לשבור את המסלול לסדרת שמות קבצים, כאשר כל השמות פרט לאחרון מייצגים מדריכים, ושימוש במטמון ה-dentry יכול להאיץ את החיפוש. יש לוודא שהתהליך מורשה לגשת לכל אחד מהמדריכים בדרך, ששם קובץ יכול להיות קישור סימבולי שמתייחס למסלול אחר ושכל כניסה בדרך עשויה להיות נקודת הרכבה של מערכת קבצים אחרת.
על מנת לבצע חיפוש במסלול, משתמש באלגוריתם שמשתמש בעיקר בכתובת אובייקט ה-dentry של המדריך הנוכחי ומצביע ל-superblock של מערכת הקבצים של המדריך הנוכחי (פרמטר mnt). בשלב האתחול קובעים את dentry ו-mnt בהתאם למדריך ההתחלתי. ההליכה במסלול מתבצעת באופן הבא: לוקחים את אובייקט ה-inode המוצבע ע"י dentry, בודקים שלתהליך הנוכחי יש הרשאות גישה ל-inode ומסתכלים על שם הקובץ הבא במסלול. אם הוא נקודה (.) מתעלמים ועוברים לקובץ הבא (. מציינת מדריך נוכחי), אם הוא שתי נקודות (..) מנסים לעלות למדריך האב המוצבע מתוך ה-dentry, אחרת מחפשים את שם הקובץ במטמון ה-dentry. אם לא מצאנו, נדרשת גישה לדיסק ויצירת dentry חדש. בודקים אם הוא מהווה נקודת הרכבה למערכת קבצים כלשהי, ואם כן מעדכנים את mnt להצביע למערכת קבצים זו ואת dentry להצביע למדריך הראשי שלה. בודקים אם ה-inode של ה-dentry החדש מייצג קישור סימבולי, בודקים שה-dentry החדש מצביע למחדריך (אם מצביע לקובץ – שגיאה), ומעדכנים את dentry החדש להיות dentry הנוכחי ועוברים למרכיב הבא. ממשיכים עד שנגיע לשם הקובץ האחרון במסלול וה-inode של ה-dentry האחרון הוא זה שחיפשנו.

פתיחת קובץ – מבוצעת ע"י קריאת המערכת open(), שמפעילה את sys_open(). במסגרתה מתבצעות קריאות לפונקציות נוספות ויתבצעו הפעולות הבאות: קוראים את המסלול לקובץ מזיכרון תהליך המשתמש, מוצאים את המקום הפנוי הראשון fd ב-PDT של התהליך, משתמשים באלגוריתם לאיתור מסלול לאיתור ה-inode המתאים למסלול (אם צריך, יוצרים אחד חדש בדיסק), יוצרים file object חדש ומעדכנים את השדות שלו בהתאם לפרמטרים, מפעילים את פונקציית open של ה- file object החדש במידה ומוגדרת, וקובעים את תוכן התא ה-fd ב-PDT להצביע ל-file object החדש.

סגירת קובץ – מבוצעת ע"י קריאת המערכת close(), שתפעיל את sys_close(). במסגרתה מתבצעות קריאות לפונקציות נוספות ויתבצעו הפעולות הבאות: מוצאים את ה-file object ב-PDT של התהליך, בתא שהאינדקס שלו fd, קובעים את ערך התא ל-NULL, מפעילים את פונקצית flush של ה- file object (אם מוגדרת, היא מבצעת כתיבה של חוצצי הקובץ בחזרה לדיסק), משחררים את ה- file object ומחזירים את קוד השגיאה של פונקציית flush (בדרך כלל 0).

קריאה וכתיבה לקובץ – מתבצעות ע"י קריאות המערכת read() ו-write(), שיפעילו את sys_read() ו-sys_write() בהתאמה. בשני המקרים יבצעו את הפעולות הבאות: מוצאים את ה-file object באמצעות ה-fd הנתון, בודקים האם הדגלים ב-file object מאפשרים את הפעולה המבוקשת, מפעילים את הפונקציה המתאימה ב-file object כדי להעביר את המידע מהחוצץ או אליו.

ניהול זיכרון

מערכת ההפעלה צריכה לנהל את השימוש בזיכרון, היות וחלק מהזיכרון מוקצה למערכת ההפעלה עצמה בעוד שאר הזיכרון מתחלק בין התהליכים שרצים כרגע ויש צורך להקצות זיכרון לתהליך שמתחיל לרוץ ולקחת בחזרה זיכרון כשתהליך מסיים. מערכת ההפעלה צריכה למנוע מתהליך גישה לזיכרון של תהליכים אחרים, כולל לזיכרון שלה עצמה. זיכרון של תהליך שאינו רץ מועבר לדיסק (swap-out), בעוד כאשר תהליך חוזר לרוץ, מקצים לו מחדש מקום ומביאים את הזיכרון שלו (swap in).

מרחב הכתובות של התהליך – בעיקרון, כל מרחב הכתובות צריך להיות זמין בזיכרון הפיזי של המחשב כאשר התהליך רץ.
עקרון הלוקליות – תהליך ניגש רק לחלק מזכרי של הזיכרון שברשותו בכל פרק זמן נתון. צריך לאחסן ערכים של משתנים, גם אם לא משתמשים בהם, אבל לא בזיכרון הפיזי.
הדיסק מכיל חלקים מהזיכרון של תהליך שרץ כרגע. צריך לזכור מה נמצא בזיכרון הפיזי ואיפה, צריך לבחור מה להעביר אליו וצריך לזכור איפה שמנו חלקי זיכרון בדיסק, כדי לקרוא אותם בחזרה, אם נצטרך אחר כך.

זיכרון וירטואלי – מספק מרחב כתובות מלא לכל תהליך, כאשר הוא יכול להיות גדול מגודל הזיכרון הפיזי, ורק חלקי הזיכרון הנחוצים כרגע לתהליך נמצאים בזיכרון הפיזי. תהליך יכול לגשת רק למרחב הכתובות שלו, כאשר מרחב הכתובות הפיזי מוגבל בגודל הזיכרון הפיזי במחשב, ואילו מרחב הכתובות הווירטואלי שאותו רואה כל תהליך אינו מוגבל בגודלו (אלא ע"י גודל הדיסק).

הפרדה בין תהליכים – נחוצה לצורך הגנה כדי לא לאפשר לתהליך גישה לנתונים של תהליך אחר ע"י כך שכתובות מתורגמות לתוך מרחב הכתובות הווירטואלי של תהליך זה בלבד, ולצורך שמירה על אי תלות בביצועים, כאשר מערכת ההפעלה מחלקת משאבים מצומצמים בין כמה תהליך, ודרישות הזיכרון (פיזי) של תהליך לא על חשבון תהליך אחר.

חלוקה קבועה – שיטה ישנה, שלפיה מערכת ההפעלה מחלקת את הזיכרון הפיזי לחתיכות בגודל קבוע שיכולות להכיל חלקים ממרחב הכתובות של תהליך. לכל תהליך מוקצית חתיכת זיכרון פיזי, כאשר מספר התהליכים הרצים קטן או שווה למספר החתיכות.

הבעיות של השיטה הזו היא שמתרחשת החלפה של חתיכת זיכרון שלמה בבת אחת, ועבודה עם כתובות זיכרון רציפות, שאם הוא גדול מספיק להכיל את כל מה שצריך הוא עלול לגרום לכך שכל החלפה תיקח הרבה זמן, ותאפשר למעט מדי חתיכות שונות, דבר שיגרום למעט מדי תהליכים לרוץ בו זמנית.
שברור פנימי – חלק מחתיכת הזיכרון של התהליך מבוזבז.

חלוקה משתנה – שיטה ישנה נוספת, שהיא הרחבה של השיטה הקודמת ע"י תוספת רגיסטר שמציין את אורך החתיכה. השיטה מונעת שברור פנימי אבל גורמת ל**שברור חיצוני**, יצירה של שאריות מקום בין החתיכות שלא מתאימות לכלום.

דפדוף
השיטה המודרנית לטיפול בזיכרון. מחלקים את הזיכרון הווירטואלי לדפים בגודל קבוע, כאשר הם גדולים מספיק לאפשר כתיבה / קריאה יעילה לדיסק וקטנים מספיק לתת גמישות (גודל טיפוסי של 4K). הזיכרון הפיזי מחולק למסגרות (frames) בגודל דף, כאשר כל מסגרת בזיכרון הפיזי יכולה להחזיק כל דף וירטואלי. כל דף וירטואלי נמצא בדיסק, וחלק מהדפים הווירטואלים נמצאים בזיכרון הפיזי.

טבלת הדפים – קיימת אחת לכל תהליך, כאשר כל כניסה בטבלה מתייחסת למספר דף וירטואלי (האינדקס של הכניסה) ומכילה מספר מסגרת פיזית (ערך הכניסה). לכתובת יש שני חלקים: מספר הדף (Virtual Page Number) ומיקום בתוך הדף (offset). ה-VPN מהווה מצביע לטבלת הדפים ומאפשר למצוא את מספר המסגרת שבו נמצא הדף (Physical Page Number), והוספת ה-offset נותנת את הכתובת עצמה.
הכניסות בטבלת הדפים מכילות גם מידע ניהולי, בנוסף למיקום הדף הפיזי:

- **Valid bit** – האם הכניסה רלוונטית. דולק כאשר הדף בזיכרון, כבוי אם הדף נמצא רק בדיסק.
- **Reference bit** – האם ניגשו לדף. דולק בכל גישה לדף, ומכובה באופן ידני ע"י מערכת ההפעלה שעוברת אחת לזמן מסוים ומכבה לא פעילים.
- **Modify bit** – האם היתה כתיבה לדף. בהתחלה כבוי, דולק כאשר יש כתיבה לדף.
- **Protection bits** – מה מותר לעשות על הדף. למשל האם הוא מכיל קוד לביצוע, האם מותר לקרוא ו/או לכתוב אליו ולמי יש הרשאות גישה אליו.

Page Fault – כאשר החומרה ניגשת לזיכרון לפי טבלת הדפים ומגיעה לדף שאינו בזיכרון הפיזי, נגרמת חריגה מסוג Page Fault. בטיפול בחריגה זו, גרעין מערכת ההפעלה טוען את הדף המבוקש למסגרת בזיכרון הפיזי ומעדכן את טבלת הדפים, וייתכן שיהיה צורך לפנות דף ממסגרת בזיכרון לצורך טעינת הדף החדש, כולל כתיבת הדף הישן לדיסק אם הוא עודכן. כאשר מסתיים הטיפול בחריגה, מבוצעת ההוראה מחדש. החריגה יכולה להיגרם גם מסיבות אחרות כמו גישה לא חוקית לזיכרון או גישה לדף לא מוקצה, כאשר שגרת הטיפול צריכה לדעת למה קראו לה.

יתרונות הדפדוף
- **חוסר שברור חיצוני** – כל מסגרת פיזית יכולה לשמש לכל דף וירטואלי ומערכת ההפעלה זוכרת איזה מסגרות פנויות.
- **מצמצם שברור פנימי** – דפים קטנים בהרבה מחתיכות.
- **קל לשלוח דפים לדיסק** – בוחרים גודל דף שמתאים להעברה בבת אחת לדיסק, ולא חייבים למחוק אלא רק ניתן לסמן כלא רלוונטי.
- אפשר להחזיר בזיכרון הפיזי קטעים לא רציפים.

- **גודל טבלאות הדפים מוגבל** – 4 בתים לכל כניסה, 2^{20} כניסות, כאשר לכל תהליך יש טבלת דפים בגודל 4MB.
- **תקורה של גישות לזיכרון** – לפחות גישה נוספת לזיכרון (לטבלת הדפים) על מנת לתרגם את הכתובת.
- **עדיין יש שברור פנימי** – גודל זיכרון התהליך אינו כפולה של גודל הדף (שאריות בסוף), ולמרות שדפים קטנים ממזערים שברור פנימי, דפים גדולים מחפים על שהות בגישה לדיסק.

טבלת דפים בשתי רמות – שולחים חלקים מטבלת הדפים לדיסק (למשל, דפים שמתייחסים לתהליכים לא פעילים) ע"י רמה נוספת של הצבעה. טבלת דפים בריבוע מאפשרת למצוא את הדפים של טבלת הדפים.
לכתובת וירטואלית יש 3 חלקים:

- **Master page** – מצביע על הכניסה ברמה העליונה, שמצביעה לטבלת הדפים המתאימה. גודלו 10 ביטים.
- **Secondary page** – המיקום בטבלת הדפים ממנו מוציאים את ה-page frame של הכתובת הפיזית. גודלו 10 ביטים.
- **Offset** – מצביע ל-offset של הכתובת הפיזית.

רצוי שהטבלה ברמה העליונה תיכנס בדף אחד וגודלה יהיה 4KB, כלומר 1024 כניסות, כ"א עם 4 בתים. היא תהיה תמיד בזיכרון הראשון הראשי. קיימת תקורה של שתי גישות לזיכרון (ולפעמים גישה לדיסק) על כל גישה לדף, והפתרון הוא שימוש במטמון (cache) קטן וזריז במיוחד.

Translation Lookaside Buffer (TLB) – מטמון חומרה ששומר מיפויים של מספר דפים וירטואליים למספר דפים פיזיים. למעשה, הוא שומר עבור כל מספר דף וירטואלי את כל הכניסה שלו בטבלת הדפים שיכולה להכיל מידע נוסף, כאשר המפתח הוא מספר הדף הווירטואלי. גודל המטמון הוא קטן מאוד (16-48 כניסות, 64-192KB) והוא מהיר מאוד היות והוא משתמש בחיפוש במקביל. אם כתובת נמצאת ב-TBL, הרווחנו, אולם אם יש החמצה יבוצע תרגום רגיל דרך טבלת הדפים.
אם תהליך מסוים ניגש ליותר זיכרון, יהיו לנו יותר החטאות ב-TLB, אולם הוא פוגע כמעט ב-99% של הכתובות. הסיבה לכך היא מידה רבה של מקומויות (locality) בגישה לזיכרון, בין אם היא לוקליות במקום (מסתובבים בכתובות קרובות) או בזמן (חוזרים לאותן כתובות בזמנים קרובים).

סגמנטים – חלוקה של זיכרון התהליך לחלקים עם משמעות סמנטית, לקוד, מחסנית, משתנים גלובליים וכו', כאשר הביטים העליונים של הכתובת הוירטואלית מציינים את מספר הסגמנט. מנהלים אותם כמו זיכרון עם חלוקה משתנה (חתיכות באורך לא קבוע), כאשר לכל סגמנט יש רגיסטר בסיס וגבול, המאוחסנים בטבלת סגמנטים. קיימת מדיניות שיתוף והגנה שונה לסגמנטים שונים, כאשר אפשר לשתף קוד אבל אסור לשתף מחסנית זמן ביצוע.
מערכות IA32 מאפשרות שילוב של סגמנטציה ודפדוף, כאשר סגמנט מוגדר כאוסף רציף של דפים וירטואליים. ב-Windows יוצרים לכל תהליך סגמנט נפרד לקוד, סגמנט נפרד לנתונים וכו', בעוד Linux אינה מנצלת את מנגנון הסגמנטציה המוצע בחומרה (בגלל שהיא עובדת גם על ארכיטקטורות אחרות שלא תומכות בסגמנטציה) אולם יש לה מנגנון סגמנטציה פנימי בצורה של אזורי זיכרון.

דפדוף לפי דרישה – מביאים דף רק אם התהליך דורש אותו (demand paging), אחרת יתרחש page fault. לעומת זאת, prepaging מנסה לנבא איזה דפים ידרשו ומביא אותם מראש (אולם יתכן ויגיעו דפים שאין בהם צורך).

מדיניות ההחלפה

המטרה העיקרית היא מזעור מספר פסיקות הדף, כאשר מחיר פינוי דף מלוכלך יקר יותר מפינוי דף נקי. הטיפול בפסיקת דף מתבצע on line, בעוד תהליך פינוי דפים מתבצע ברקע (off line) כאשר מספר המסגרות הפנויות יורד מתחת לאיזשהו סך מסוים. דפים מלוכלכים נכתבים ברקע לדיסק.

אלגוריתם FIFO – מפנה את הדף שנטען לפי הכי הרבה זמן. יכולה להיווצר כאן אנומלית Belady.

אלגוריתם מחסנית – לכל סדרת גישות לזיכרון, הדפים שהיו בזיכרון כאשר מספר המסגרות הוא n, מהווים תת קבוצה של הדפים שהיו בזיכרון כאשר יש n+1 מסגרות.

אלגוריתם אופטימלי – אם כל הגישות לזיכרון היו ידועות מראש, היה ניתן להשתמש באלגוריתם חמדן שמפנה מהזיכרון את הדף שהזמן עד לגישה הבאה אליו הוא הארוך ביותר.

אלגוריתם Least Recently Used (LRU) – מפנה את הדף שהגישה האחרונה אליו היא המוקדמת ביותר (למעשה יכול להתנהג כמו החמדן או כמו FIFO). המימוש שלו משתמש בחותמת זמן לכל דף, כאשר בכל גישה לדף מעדכנים את החותמת, ומפנים את הדף עם הזמן המוקדם ביותר. המימוש מחייב ניהול מחסנית, כאשר בגישה לדף מעבירים את הדף לראש המחסנית, ומפנים את הדף בתחתית המחסנית. המימוש הוא יקר ודורש תמיכה בחומרה.

אלגוריתם ההזדמנות השניה (LRU מקורב) – לכל דף מוסיפים reference bit וכאשר ניגשים לדף, מדליקים את הביט. בפינוי דפים, מערכת ההפעלה עוברת באופן מחזורי על כל הדפים, ואם הביט כבוי, הדף מפונה, אחרת מאפסים את הביט.

שיבה טובה (LRU מקורב) – מצרפים שדה גיל לכניסה של דף בטבלת הדפים. דפים שלא ניגשו אליהם הרבה זמן מפונים מהזיכרון.

מדיניות הדפדוף ב-Windows NT – לפי דרישה, אבל מביאים קבוצת דפים מסביב. אלגוריתם הדפדוף הוא FIFO על הדפים של כל תהליך בנפרד. מנהל הזיכרון הווירטואלי עוקב אחרי ה- working set של התהליכים ומריץ רק תהליכים שיש בזיכרון מקום לכל ה-working set שלהם, כאשר גודלו של ה-working set נקבע עם התחלת התהליך, לפי בקשתו ולפי הצפיפות הנוכחית בזיכרון.

מדיניות דפדוף ב-Linux 2.4 – גם כאן לפי דרישה והבאת קבוצת דפים מסביב. אלגוריתם הדפדוף הוא גלובלי והוא קירוב ל-LRU (LFU – מפנים את הדף שהכי פחות השתמשו בו, האלגוריתם מנסה לשלב בין שניהם). הדיסק מופעל כאמצעי אחרון בלבד, ודפים מפונים רק כאשר הזיכרון הפיזי מתמלא מעבר לסף עליון, ומפנים עד מתחת לסף התחתון. הזיכרון המשמש את הגרעין אינו מדופדף כלל לדיסק וממופה לאותו חלק של הזיכרון הווירטואלי של כל התהליכים.

תרגום כתובות בחלוקה קבועה – תרגום כתובת וירטואלית לכתובת פיזית מתבצע באמצעות רגיסטר $base_P$ שמחזיק את כתובת הבסיס הפיזית של התהליך שרץ במעבד, ורגיסטר $base_V$ שמציין את כתובת הבסיס של החלק ממרחב הכתובת של התהליך שרץ, שנמצא כרגע בזיכרון הפיזי. כתובת וירטואלית va נמצאת בזיכרון הפיזי אם היא בתחום של $[base_V, base_P + SIZE)$, אחרת צריך להביא את החתיכה המתאימה מהדיסק (במידה וקיימת). כתובת פיזית = $base_P + (va - base_V)$.

החלפת דפים עם חלוקה קבועה – בהחלפת חתיכות זיכרון של אותו תהליך, מציבים ברגיסטר $base_V$ את כתובת החלק הווירטואלי החדש שנטען לזיכרון. בהחלפת תהליך, מציבים ברגיסטר $base_P$ את כתובת התחלת החתיכה הפיזית שבה נמצא הזיכרון של התהליך החדש.

ניהול זיכרון ב-Linux

הזיכרון הפיזי הוא זיכרון הנמצא בהתקנים הפיזיים של המחשב, והוא מורכב מזיכרון פיזי ראשי (RAM, אלקטרוני) וזיכרון פיזי משני (דיסק קשיח). זיכרון וירטואלי הוא מרחב זיכרון מדומה העומד לרשות תהליך, והוא ממופה בחלקו לזיכרון הפיזי הראשי ובחלקו לזיכרון הפיזי המשני. הקוד המבוצע ע"י המעבד והנתונים הדרושים לביצוע הקוד חייבים להיות בזיכרון הפיזי הראשי בזמן הביצוע.

<u>המטרות העיקריות של זיכרון וירטואלי</u> –

1. **הפרדה בין תהליכים** – לכל תהליך מוגדר מרחב זיכרון וירטואלי משלו, בלתי תלוי בתהליכים אחרים, ובכל נקודת זמן רק חלק ממרחב הזיכרון של תהליך נמצא בזיכרון הפיזי הראשי.
2. **הגדלת הזיכרון העומד לרשות המערכת** – מאפשר למחשב להריץ מספר רב של תהליכים שסך כל הזיכרון שלהם גדול מהזיכרון הפיזי הראשי (או תהליך בודד).

לכל תהליך מרחב זיכרון וירטואלי משלום, מתחיל בכתובת 0 ומסתיים בכתובת 3GB-1. מרחב הזיכרון של תהליך מחולק לדפים, שהם קטעי זיכרון עוקבים בעלי גודל קבוע (בד"כ 4KB). רק חלק קטן ממרחב הזיכרון של תהליך מוקצה לו עם היווצרותו, והוא יכול להגדיל את מרחב הזיכרון ע"י בקשות להקצאת זיכרון.

<u>אזורי זיכרון</u> – מרחב הזיכרון של תהליך מתחלק לאזורים, שהם רצפים נפרדים של כתובות במרחב הזיכרון שרק אליהם התהליך יכול לגשת, תחת הראשות ספציפיות לכל אזור, והם קובעים תת תחומים של כתובות שנמצאים בשימוש התהליך. כתובת התחלתית וגודל של אזור זיכרון הם כפולות של גודל הדף. ניתן להוסיף, להסיר, להגדיל ולהקטין אזורי זיכרון וניתן לשתף אזור זיכרון בין מספר תהליכים.
לכל אזור זיכרון קיים מתאר אזור זיכרון שכולל את כתובת ההתחלה שלו, כתובת אחת אחרי האחרונה ודגלים המציינים תכונות של האזור וכוללים את ההרשאות שלו (קריאה, כתיבה, ביצוע שיתוף) והן מאפשרות למערכת ניהול הזיכרון לזהות גישות חוקיות לא חוקיות לדפים באזור.
כשתהליך נוצר, הוא מקבל מרחב זיכרון עם קבוצה של אזורי זיכרון סטנדרטיים למערכות UNIX (אזור לקוד, לנתונים סטטיסטיים, לערימה של הזיכרון הדינמי, למחסנית user mode) ואזורים נוספים (אחד לפרמטרים של שורת הפקודה, אחד למשתני מערכת). במהלך הריצה, קבוצת אזורי הזיכרון של התהליך יכולה להשתנות, כאשר אזור המחסנית ב-user mode יכולה לגדול (אך לא לקטון), אזור הערימה יכול לגדול או לקטון לפי הצורך ומיפוי קבצים, הפעלה או סגירה של מנגנוני זיכרון משותף גורמים להוספה והסרה של אזורי זיכרון למרחב זיכרון של תהליך.
ניתן למפות קובץ נתונים ממערכת הקבצים לאזור זיכרון של תהליך. גישה ל-byte כלשהו בדף השייך לאזור הזיכרון מתורגמת לגישה לאותו byte בתוך הקובץ.
מרחב הזיכרון יכול להיות משותף למספר תהליכים, וזה שימוש בעיקר למימוש חוטים. מרחב זיכרון משותף מפונה כאשר אינו בשימוש ע"י אף תהליך.

<u>ניהול דפים</u> – מרחב הזיכרון של תהליך מחולק לדפים, כאשר רק חלק מהם נמצאים בזיכרון הפיזי הראשי בכל זמן נתון (אלה שהשתמשו בהם לאחרונה). הזיכרון הפיזי הראשי של המחשב מחולק למסגרות, כל אחת מהם בגודל דף. דף שנמצא בזיכרון הראשי משובץ למסגרת כלשהי.

<u>טבלת הדפים</u> – לכל תהליך יש טבלת דפים משלו שמנהלת את הדפים שלו (האם הדף נמצא בזיכרון הראשי ובאיזה מסגרת). הטבלה מכילה כניסה עבור כל דף במרחב הזיכרון של תהליך, כאשר כל כניסה של דף בטבלת הדפים מכילה דגל מיוחד המציין האם הדף נמצא בזיכרון הראשי, ואם כן, היא מכילה את מספר המסגרת בה מאוחסן הדף בזיכרון הראשי (אם לא, היא מצביעה על המיקום של הדף בדיסק).
הגישה לנתונים נעשית לפי כתובת ליניארית בגודל 32 ביט. גישה של תהליך לכתובת ליניארית

מתורגמת לגישה לכתובת פיזית בזיכרון הראשי, ובתרגום שלה המעבד מחשב באיזה דף נמצאת הכתובת הליניארית ואת מרחקה מתחילת הדף והוא בודק בטבלת הדפים באיזה מסגרת נמצא הדף. הכתובת הפיזית היא כתובת תחילת המסגרת + offset.
אם הדף המכיל את הכתובת אינו נמצא בזיכרון, נוצרת חריגה מסוג page fault שגורמת לגרעין להביא את הדף לזיכרון, ובסיום הטיפול בפסיקה מבוצעת מחדש ההוראה שגרמה לפסיקה.
מנגנון הדפדוף של Linux מביא דפים לפי דרישה, ומקצים לו מסגרת רק בעקבות page fault הנגרם ע"י ניסיון גישה לדף. המסגרת שמוקצית לדף נלקחת ממאגר של מסגרות פנויות, דבר שעלול לגרום לירידת מספר המסגרות הפנויות.

<u>מאגרי דפדוף ב-Linux</u> – מאגר דפדוף (swap area) הוא אזור מיוחד בדיסק אליו מפונים דפים מהזיכרון הפיזי הראשי. Linux מאפשרת להגדיר מספר מאגרי דפדוף, כך שניתן להפעיל ולכבות מאגרי דפדוף באופן דינמי תוך כדי פעולת המערכת, ועומס הדפים המפונים מחולק בין המאגרים. כל מאגר דפדוף הוא שטח דיסק המחולק למגירות, כשכל מגירה משמשת לאחסון דף מהזיכרון הראשי וגודלה כגודל דף. המגירה הראשונה מכילה מידע ניהולי על המאגר כמו גודל, גרסת אלגוריתם דפדוף וכו'. אלגוריתם הדפדוף משתדל להקצות מגירות ברצף לדפים מפונים כדי לשפר את זמן הגישה, ואילו דפים הממופים לקבצים מועברים לקובץ ממנו באו כאשר הם מפונים מהזיכרון הפיזי הראשי.

<u>מתאר הזיכרון של תהליך</u> – מתאר זיכרון מכיל מידע רב על מרחב הזיכרון. השדה mm במתאר תהליך מצביע על מתאר הזיכרון של התהליך, כאשר מתארי תהליכים שחולקים את אותו מרחב זיכרון, מצביעים על אותו מתאר זיכרון, וערך שדה mm של תהליך גרעין הינו NULL. כל מתארי הזיכרון שבמערכת משורשרים ברשימה מקושרת.

<u>טבלאות הדפים</u> – לכל תהליך יש טבלת דפים עם כניסה עבור כל דף במרחב הזיכרון של התהליך, וקיים רגיסטר מיוחד (cr3) שמצביע על טבלת הדפים של התהליך הנוכחי, ובכל context switch משנים את ערך הרגיסטר. תהליך מנצל בדרך כלל רק חלק מזערי ממרחב הזיכרון הווירטואלי ולכן לא צריך להחזיק את הטבלה כולה באופן זמין, וכדי לפתור את זה מחזיקים בשתי רמות היררכיה (או יותר) בטבלה. אם מוקצה דף חדש לשימוש התהליך, צריך להקצות לו לפי הצורך דפים נוספים לטבלאות ביניים בהיררכיה עד השורש (לא כולל).

<u>מבנה כניסה בטבלת הדפים</u> – כניסה בטבלת הדפים היא בגודל 32 bit, והמידע שהיא מכילה תלוי בביט present המציין האם הדף נמצא בזיכרון הראשי.
אם ביט ה-presnet דולק, 20 ביטים בכניסה מציינים את מספר המסגרת בה מאוחסן הדף, ביט accessed מודלק ע"י החומרה בכל פעם שמתבצעת גישה לכתובת בדף, ביט dirty מודלק ע"י החומרה כל פעם שמתבצעת כתיבה לנתון בדף (אם הוא דלוק, יש לשמור עליו ולא לזרוק אותו), ביט read/write שמציין הרשאת גישה (כבוי לקריאה בלבד, דלוק לקריאה וכתיבה, כאשר הרשאות הדף נקבעות לפי הרשאות האזור) וביט user/supervisor שמציין גישה מיוחסת ודולק כאשר יש גישה לכל תהליך (כבוי – גישה לקוד הגרעין בלבד).
אם ביט ה-present כבוי, הדף נמצא במאגר דפדוף (swap) ושומרים עבורו מזהה מגירה שמורכב ממספר מאגר הדפדוף בו נמצא הדף (קיימים 128 מאגרים שונים בזיכרון הפיזי) ומספר המגירה במאגר הדפדוף שבה הדף נמצא.
מתאר אזור זיכרון אינו ניתן לשיתוף, ולכן אם לשני תהליכים או יותר יש אזור זיכרון משותף, יהיה לכל אחד מהם מתאם אזור זיכרון משלו שמצביע לרצף הדפים המשותף. עבור כל דף באזור הזיכרון המשותף יש כניסה בטבלת הדפים של כל אחד מהתהליכים, והיא מצביע על אותה מסגרת המכילה את הדף. הכניסות המצביעות על אותו דף בטבלאות הדפים של תהליכים ושנים יכולות להכיל הרשאות שונות.

תהליך בן יכול להיווצר כשותף למרחב הזיכרון של האב (כשיוצרים חוטים, ואז רק מגדילים את מונה השיתוף של מתאר הזיכרון של תהליך האב) או שתהליך הבן יקבל מרחב זיכרון משלו (למשל בקריאת ()fork ואז צריך להעתיק את מרחב הזיכרון של האב לזה של הבן.

העתקת מרחב זיכרון – העתקה פשוטה של מרחב הזיכרון תהיה יקרה (דורשת הרבה זמן) ואולי מיותרת (מרחב הזיכרון של תהליך הבן יאותחל מחדש אם הוא יטען תוכנית חדשה מיד עם תחילת ביצועו). Linux משתמשת ב-**Copy On Write (COW)** שלפיה דפים הניתנים לכתיבה שאינם יכולים להיות משותפים (דפי נתונים ומחסנית) מוגדרים בתחילה כמשותפים אבל מועתקים לעותק פרטי כאשר אחד התהליכים השותפים (האב או הבן) מנסה לכתוב אליהם לראשונה, בעוד שאר הדפים הופכים למשותפים בין מרחב הזיכרון של האב והבן. החיסרון בטכניקה הוא שלאחר ()fork, כתיבה ראשונה לכל דף שאינו משותף יקרה בגלל הטיפול בחריגות דף.
כש-COW יוצרת מרחב זיכרון חדש לבן, היא יוצרת (באמצעות הפונקציה ()copy_mm) עבורו עותק של מתאר הזיכרון של תהליך האב. לכל אזור זיכרון של האב מתאר אזור הזיכרון מועתק למתאר אזור זיכרון חדש של הבן, הכניסות בטבלאות הדפים הממפות את האזור מועתקות לכניסות בטבלאות הבן וכל הדפים הופכים למשותפים, דפים ששייכים לאזור שלא ניתן לשיתוך וניתן לכתיבה יסומנו בטבלת הדפים של האב והבן כדפים לקריאה בלבד ואילו עבור דפים שמשותפים היא תגדיל את מונה השיתוף במסגרת.
כאשר האב או הבן כותבים לדף, יש חריגת page fault. המערכת תקבע אם לשכפל את הדף לפי ערך count, כאשר אם הוא גדול מ-1 היא תשכפל אותו לעותק חדש במסגרת מוקצית אחרת (בישנה יוקטן המונה ובחדשה יוצב 1 ותתאפשר הכתיבה) ואם הוא בדיוק 1, הגרעין יאפשר כתיבה בדף.

טיפול ב-TLB – Translation Lookaside Buffer הוא מטמון הצמוד למעבד המכיל כניסות בטבלת הדפים על מנת לחסוך תרגומים חוזרים של אותה כתובת ליניארית לכתובת פיזית, דבר החוסך גישות לזיכרון לצורך תרגום. גרעין Linux פוסל כניסות ב-TLB כשמתבצע עדכון של רשומה בטבלת הדפים שמופיעה גם ב-TLB. בכל החלפת הקשר מתבצעת פסילה אוטומטית של תוכן ה-TLB, אלא אם ההחלפה היא בין חוטים, כאשר שני התהליכים משתפים את אותו מרחב זיכרון או כאשר התהליך הבא לביצוע הוא תהליך גרעין, שאין לו מרחב זיכרון משלו (פועל על מרחב הזיכרון של הגרעין ומנצל את טבלאות הדפים של תהליך המשתמש שרץ לפניו).

מרחב זיכרון גרעין – מרחב הזיכרון של מעבד הוא 4GB, וממנו הגרעין משתמש במרחב זיכרון בגודל 1GB. מרחב זה אינו מופיע ברשימת אזורי הזיכרון של תהליך, והגרעין שומר בו את מבני הנתונים המתארים את מרחבי הזיכרון של כל התהליכים ואת טבלאות הדפים של כל התהליכים (כדי שלא נצטרך לעשות flush ל-TLB בכל גישה לזיכרון. הוא לעולם לא מפונה לדיסק.
טבלת הדפים למרחב הזיכרון של הגרעין נקראת Kernel Master Page Global Directory והיא מתעדכנת כל פעם שהגרעין מקצה ומשחרר דפים לשחרור עצמו בלבד. היא משמשת כמקור ממנו מתעדכנות טבלאות הדפים של תהליכי המשתמש לגבי דפים שבשימוש הגרעין, והיא אמורה לפתור בעיה של page fault אם שינינו מקום ב-1GB של הגרעין ב-PGD של תהליך מסוים ולא שינינו אותו אצל התהליכים האחרים.

טיפול ב-page fault ב-Linux – החומרה מתריעה באמצעות page fault על גישה לדף שאינו נמצא בזיכרון או גישה לא חוקית (לא לפי ההרשאות בטבלת הדפים) לדף שנמצא בזיכרון. הגרעין צריך לנתח את נסיבות החריגה ולהחליט אם היא חוקית וכיצד לטפל בה. החומרה מעבירה לשגרת הטיפול קוד שגיאה של 3 ביטים הנשמר במחסנית כאשר אם ביט 0 כבוי הגישה היא לדף שאינו בזיכרון (אחרת גישה לא חוקית לדף בזיכרון), אם ביט 1 כבוי הגישה היתה לקריאה או לביצוע קוד (אחרת לכתיבה) ואם ביט 2 כבוי, הגישה היתה כשהמעבד היה ב-Kernel Mode (אחרת, ב- User Mode). ערך הכתובת הווירטואלית שגרמה לחריגה נשמר ברגיסטר cr2.

אם הגישה לכתובת בגרעין בדך לא קיים במצב kernel mode, הגרעין יקצה לעצמו דפים, יעדכן רק את הטבלאות המרכזיות שלו, יקשר את כל רמות ההיררכיה החסרות (מה-PGD ומטה) בטבלת מרחב הזיכרון הנוכחי לאלה שבטבלאות הגרעין (אם אין אובייקטים מתאימים באחת הרמות בטבלה המרכזית של הגרעין, הגישה שגויה ומקבלים הודעת תקלה) ויעדכן את טבלאות הדפים של מרחבי זיכרון של תהליכי משתמש לגבי דפים שבשימוש הגרעין. אם הגרעין משחרר דפים (ומעדכן את הטבלאות שלו בהתאם), השינוי משתקף מיידית בכל המרחבים האחרים, כאשר הגרעין לעולם לא משחרר את האובייקטים בתוך הטבה, והכניסות (מתחת ל-PGD) בטבלת הדפים של מרחב זיכרון מקושרות לאובייקטים שבתוך הטבלה של הגרעין ולא להעתקים שלהם.

דפדוף לפי גישה – אם הכתובת המבוקשת נמצאת בתוך אחד מאזורי הזיכרון, הגישה בהתאם להרשאות והדף המבוקש אינו בזיכרון, יש לטעון את הדף המבוקש לזיכרון. הדף יכול שלא להיות בזיכרון אם הכניסה מכילה ערך שאינו NULL, כלומר מזהה דף פיזי של דף ממופה אנונימי (נמצא במאגר דפדפוף) או שהכניסה מכילה ערך NULL, ואז דף "קר" כלומר ממופה אנונימי שהתהליכים החולקים את מרחב הזיכרון מעולם לא ניגשו אליו או לא כתבו אליו, ואז אם הגישה לכתיבה מקצים לו מסגרת חדשה שממולאת באפסים ואם הגישה לקריאה הכניסה בטבלת הדפים מצביעה על מסגרת של דף קבוע מיוחד שממולא באפסים.

טיפול בתקלות – החריגה נקבעת כתקלה (גישה לא חוקית) אם הפעולה לא מורשית לפי הרשאות האזור, גישה מקוד משתמש לדפי הגרעין, גישה לכתובת בתחום המשתמש שאיננה בתוך אזור זיכרון (אלא אם התבצעה כתיבה למחסנית). אם הגישה היתה מקוד תהליך משתמש, נשלח לו signal שמציין גישה לא חוקית לזיכרון, ואם הגישה היתה מקוד גרעין, מוכרזת תקלת מערכת.

למה צריך את מטמון הדפים – דף יסומן שהוא לא בזיכרון באחת משתי האופציות הבאות:

1. **לפני הפינוי בפועל** ואז כשיוחלט לפנות דף קודם כל הוא יסומן ע"י present=0 וכשהתהליך ינסה לגשת לדך הוא ייגש לזיכרון המשני לפי הכתובת במאגר הדפדוף. המערכת לא הספיקה לכתוב את הדף למאגר הדפדוף ולכן התהליך יקרא דף לא נכון (נתונים לא מעודכנים או זבל).
2. **אם אחרי הפינוי בפועל** ואז כשמתחילים לפנות את הדף, התהליך ייגש לדף כאשר present=1, הדף פונה ודף אחר נטען לאותה מסגרת ואז התהליך ייגש למסגרת המצוינת בטבלת הדפים וקיבלנו גישה לא חוקית.

יש צורך במנגנון שינהל את הפינוי וההבאה של דפים מהזיכרון הראשי ואליו, ומטמון הדפים הוא המנגנון המרכזי לטעינה ופינוי של דפים בין הזיכרון לדיסק ב-Linux. הוא ממומש כחלק מטבלת המסגרות.

טבלת המסגרות – הגרעין של Linux מחזיק מערך עם כניסה לכל מסגרת בזיכרון הראשי של המחשב. כל כניסה מכילה מספר שדות חשובים:

- **Count** – מונה שימוש של הדף המסגרת ומייצג כמה מרחבי זיכרון מצביעים לדף כשהוא בזיכרון (ערך 0 מייצג מסגרת פנויה).
- **Flags** – דגלים המתארים את מצב הדף שבמסגרת כמו נעילה עקב פעולת פינוי או הבאה שעוד לא הסתיימה, ציון שתוכן המסגרת מלוכלך ועוד.
- **Mapping** – מצביע לאובייקט ניהול המכיל מידע ופונקציות לטיפול בדף שבמסגרת לפי סוג המיפוי שלו (לקובץ או אנונימי).
- **Index** – מציין את המיקום הפיזי של הדף במאגר בדיסק (עבור מידע מקובץ את ה-offset מתחילת הקובץ, ועבור דף מזיכרון תהליך את מזהה המגירה).

מטמון הדפים – מכיל hash table המתרגם צירוף של mapping + index לכתובת מסגרת (אם יש כזו) המכילה את הדף במיקום index של האובייקט mapping. כל המסגרות שמתאימות לאותו hash מקושרות ברשימה כפולה מעגלית ברשומת המסגרת.
בדומה למסגרת, לכל מגירה במאגר הדפדוף יש מונה שימוש הסופר כמה מרחבי זיכרון מצביעים לדף המאוחסן במגירה. מבחינת מונה השימוש של מסגרת או מגירה, מטמון הדפים נחשב כמרחב זיכרון נפרד המשתמש בדף המאוחסן בה. כאשר מסגרת או מגירה הם בשימוש מטמון הדפים, מוגדל המונה ב-1, וכאשר מנתקים את המסגרת הוא המגירה מהמטמון, הוא מוקטן ב-1. המטרה היא למנוע שיבוש במיפוי ה-hash table באמצעות הקצאה מחדש של המסגרת/מגירה עד להוצאת הדף מהמטמון.

הקשר בין מסגרת למגירה – הקשר בין מסגרת ומגירה המכילות את אותו דף ממופה אנונימי הוא דינמי, כאשר דף המצוי במגירה יכול להיטען לכל מסגרת פנויה שתוקצה בעת הצורך, ודף מפונה למגירה בדיסק (והמסגרת מתפנה) רק לאחר שכך מרחבי הזיכרון השותפים בו מצביעים על המגירה בדיסק, והקשר הקיים בין המסגרת למגירה ניתן לניתוק.
הקשר הדינמי מתאפשר גם בכיוון ההפוך, כאשר לאחר שדף נטען לזיכרון וכל מרחבי הזיכרון מצביעים למסגרת, הגרעין בוחר במקרים מסוימים לשחרר את המגירה ולנתק את הקשר הקיים, ולפני שכותבים דף לדיסק, מקצים לו מגירה חדשה אם אין לו. מטמון הדפים מחזיק את הקשר בין המגירה והמסגרת של אותו דף ממופה אנונימי, וכל עוד קשר זה מתקיים. כאשר הקשר ניתק, המסגרת מוצאת ממטמון הדפים וגם המגירה.

שלבי פינוי דף מהמטמון – מטמון הדפים מצביע למסגרת בזיכרון הראשי שבתוכה נמצא הדף (ה-count של המסגרת מוגדל ב-1). מוקצית מגירה במאגר הדפדוף אליה יעבור הדף המפונה, כאשר גם המטמון מצביע למגירה זו. הדף נכתב למגירה במאגר הדפדוף, ובאופן סימטרי מבוצעת גם הבאת דף ממאגר הדפדוף לזיכרון הראשי.

טעינת דף למטמון – בטעינת דף מתבצעות הפעולות בסדר הפוך בקירוב, תוך שימוש במטמון הדפים. בטעינת דף למרחב זיכרון מסוים בודקים אם הדף כבר נטען למסגרת בזיכרון, ואם כן רק מעדכנים את טבלת הדפים ומוני השיתוף (אחרת טוענים את הדף לזיכרון ומכניסים אותו למטמון הדפים). טעינת דף משותף מתבצעת בצורה הדרגתית תוך שמירה על תיאום בין התהליכים, כאשר הדף נטען פעם אחת בלבד ולמסגרת אחת בלבד.

פינוי דפים ב-Linux

מנגנון הפינוי מורכב מ-3 מרכיבים:

דירוג דינמי של רמת הפעילות של כל דף במסגרת בזיכרון – הגרעין מחזיק 2 רשימות מקושרות כפולות מעגליות של רשמות מסגרת במטמון הדפים:

- **active_list** – רשימת מסגרת הדפים הפעילים, כלומר דפים שניגשו אליהם לאחרונה.
- **Inactive_list** – רשימת מסגרת הדפים הלא פעילים, שלא ניגשו אליהם זמן מה.

הרשימות לא חופפות, ורשומת כל מסגרת הניתנת לפינוי מקושרת לאחת הרשימות ומסגרת מוספת לרשימה דרך ראש הרשימה. הקישור של מסגרת לאחת הרשימות הוא באמצעות השדה lru ברשומת המסגרת, כאשר הדגל PG_lru דולק ברשומת מסגרת הנמצאת באחת מהרשימות. הדגל PG_active דולק רק בכל רשומת מסגרת השייכת לרשימה active_list, ואילו הדגל PG_referenced ברשומת מסגרת מציין שבוצעה גישה לדף במסגרת זו.
פינוי בפועל של דפים ממסגרות מבוצע החל מסוף רשימת ה-inactive_list.

כאשר עולה הצורך בפינוי בפועל של זיכרון – מעבר על רשימת המסגרות בעלות הפעילות הנמוכה ביותר ופינוי בפועל של המסגרות שאינן בשימוש מרחבי זיכרון של תהליכים, וזוהי בעצם פעולה של

חוסר ברירה. Linux נוטה לנצל את הזיכרון הראשי ככל האפשר מבלי לפנות דפים כלל, ומוגדר סף קריטיות של כמות מינימלית של מסגרות שחייבת להישאר פנויה. הסף הזה מוגדר לצורך הפעלת אלגוריתמים של פינוי זיכרון שצריכים מספר מסגרות פנויות, וכדי להקטין את זמן הטיפול ב- page fault שמצריך טעינת דף מהזיכרון המשני, כי אין צורך לפנות מסגרת אלא רק לטעון דף למסגרת פנויה. מוגדר גם סף עליון שמעבר לו אין צורך לבצע פינוי מסגרות ופונקצית הפינוי תנסה להגיע אליו.
מנגנון הפינוי, הקרוי מנגנון מחזור מסגרות, מופעל בקריאה לפונקציה (try_to_free_pages, שהפעלתה מבוצעת במקרה שחוט גרעין מיוחד (kswapd) מגלה שכמות המסגרות הפנויות קטנה או שווה לסף הנמוך או כאשר הקצאת מסגרת חדשה נכשלת בעקבות הגעה לסף הקריטי ואז מתבצעת הפעלה ישירה של הפונקציה או שמעירים את kswapd. הטקטיקה לשחרור מסגרות היא לבצע מספר איטרציות של ניסיונות של שחרור מסגרות, כאשר בכל איטרציה מנסים לשחרר מסגרות ע"י קריאה לפונקציות לצמצום מטמונים שונים, מנסים להגדיל את ה-inactive_list ולאחר מכן סורקים אותה ומשחררים מסגרות הנמצאות בשימוש מטמון הדפים בלבד, ומסגרת המכילה דף מלוכלך (עבר עדכון) נכתבת לדיסק לפני פינוייה.

<u>אם הגרעין מעריך שצריך לפנות דפים נוספים</u> – מתבצע מעבר על טבלאות הדפים של כל מרחבי הזיכרון של תהליכי המשתמש, וכאשר מאותרת כניסה השייכת לאזור זיכרון ומצביעה למסגרת הניתנת לפינוי ובעלת פעילות נמוכה, מבוצע פינוי של הדך ממרחב הזיכרון הנבדק.
כאשר סורקים את המסגרות ב-(inactive_list ונספרות יותר מדי מסגרות (מעבר לסף מוגדר) המסומנות בשימוש ע"י תהליכים ו/או שאינן במטמון הדפים, מופסקת הסריקה ומופעלת הפונקציה (swap_out לפינוי דפים ממרחבי זיכרון. היא מנסה לפנות דפים ממרחבי הזיכרון הטעונים למסגרת במטרה להביא את המסגרת למצב פנוי (count==0) או שהן בשימוש מטמון הדפים בלבד (count==1, mapping!=NULL), כלומר מסגרת לפני שחרור. הפונקציה סורקת את כל טבלאות הדפים הממפות את כל אזורי הזיכרון בכל מרחבי הזיכרון במערכת (עד לכמות הנסרקת), ולכל כניסה המצביעה למסגרת בזיכרון היא בודקת אם הביט accessed בכניסה בטבלה דלוק, מכבה אותו ועוברת למסגרת הבאה. הדף במסגרת ניתן לפינוי אם אזור הזיכרון המכיל את הדף ניתן לפינוי והמסגרת לא ב-active_list.