ה א ו נ י ב ר ס י ט ה ה פ ת ו ח ה

20594

מערכות הפעלה

חוברת הקורס – סתיו 2016א

כתב: דוד שריאל

אוקטובר 2015 – סמסטר סתיו- תשע״ו

פנימי – לא להפצה.

תוכן העניינים

אל הסטודנט,

אנו מקדמים את פניך בברכה עם הצטרפותך אל הלומדים בקורס " מערכות הפעלה".

בחוברת זו תמצא את לוח הזמנים, תנאים לקבלת נקודות זכות ומטלות.

לקורס קיים אתר באינטרנט בו תמצאו חומרי למידה נוספים, אותם מפרסם/מת מרכז/ת ההוראה.
בנוסף, האתר מהווה עבורכם ערוץ תקשורת עם צוות ההוראה ועם סטודנטים אחרים בקורס.
פרטים על למידה מתוקשבת ואתר הקורס, תמצאו באתר שה"ם בכתובת:
http://telem.openu.ac.il
מידע על שירותי ספרייה ומקורות מידע שהאוניברסיטה מעמידה לרשותכם, תמצאו באתר הספריה באינטרנט www.openu.ac.il/Library.

אפשר לפנות אלי בדואר אלקטרוני davidsa@openu.ac.il או בשעות הנחיה הטלפונית המפורסמות באתר הקורס. הפרטים הללו מצויים גם באתר המחלקה למדעי המחשב telem.openu.ac.il/cs. פגישות יש לתאם מראש.

חשוב להדגיש כי התקשוב בקורס ישמש ערוץ רשמי בין צוות ההוראה של הקורס לבין הסטודנט, כלומר חובה על כל סטודנט להתעדכן באופן שוטף על הנעשה בקורס דרך אתר הבית. כל ההודעות - הן בנושאים אקדמיים והן בנושאים מנהליים - יועברו דרך אתר הבית בלבד, ולא יישלחו הודעות בדואר רגיל. סטודנטים אשר אין להם גישה לרשת האינטרנט יוכלו לגשת למרכז הלימוד הקרוב לביתם ולהשתמש במעבדת המחשבים שם. לפרטים מלאים על מרכזי הלימוד ושעות הפתיחה, ניתן להתקשר למוקד הפניות בטלפון: 09-7782222.

אל אתר הבית של הקורס ניתן לגשת מדף הבית של החטיבה למדעי המחשב:
http://telem.openu.ac.il/cs

בברכת לימוד פורה ומהנה,

דוד שריאל
מרכז ההוראה בקורס

1. לוח זמנים ופעילויות (20594 /א2016)

שבוע הלימוד	תאריכי שבוע הלימוד	יחידת הלימוד המומלצת	מפגשי ההנחיה*	תאריך אחרון למשלוח הממ״ן (למנחה)
1	23.10.2015-18.10.2015	הכרת UBUNTU יחידה 1 מספר הקורס		
2	30.10.2015-25.10.2015	יחידה 2		
3	6.11.2015-1.11.2015	יחידה 2 קריאה של יחידות 10.3 ו 11.4 מספר הקורס (באופן עצמאי)		
4	13.11.2015-8.11.2015	יחידה 2		
5	20.11.2015-15.11.2015	יחידה 6		
6	27.11.2015-22.11.2015	יחידה 6		ממ״ן 11 26.11.2015
7	4.12.2015-29.11.2015	יחידה 3		
8	11.12.2015-6.12.2015 (ב-ו חנוכה)	יחידה 3		
9	18.12.2015-13.12.2015 (א-ב חנוכה)	יחידה 3		

** התאריכים המדויקים של המפגשים הקבוצתיים מופיעים ב״לוח מפגשים ומנחים״.*

לוח זמנים ופעילויות - המשך

שבוע הלימוד	תאריכי שבוע הלימוד	יחידת הלימוד המומלצת	מפגשי ההנחיה*	תאריך אחרון למשלוח הממ"ן (למנחה)
10	25.12.2015-20.12.2015	יחידה 3 קריאה של יחידות 10.4 ו 11.5 מספר הקורס (באופן עצמאי)		ממ"ן 12 24.12.2015
11	1.1.2016-27.12.2015	יחידה 4		
12	8.1.2016-3.1.2016	יחידה 4 קריאה של יחידות 10.6 ו 11.8 מספר הקורס (באופן עצמאי)		
13	15.1.2016-10.1.2016	יחידה 5		
14	22.1.2016-17.1.2016	יחידה 5 קריאה של יחידות 10.5 ו 11.7 מספר הקורס (באופן עצמאי)		ממ"ן 13 21.01.2016
15	29.1.2016-24.1.2016	יחידה 9 ושיעור חזרה קריאה של יחידות 10.7 ו 11.9 מספר הקורס (באופן עצמאי)		

מועדי בחינות הגמר יפורסמו בנפרד

** התאריכים המדויקים של המפגשים הקבוצתיים מופיעים ב"לוח מפגשים ומנחים".*

2. תיאור המטלות

קרא היטב עמודים אלו לפי שתתחיל לענות על השאלות

חוברת זו מכילה מידע על המטלות ואת המטלות עצמן.
פתרון המטלות הוא חלק בלתי נפרד מלימוד הקורס - הבנה מעמיקה של חומר הלימוד דורשת תרגול רב. המטלות יבדקו על-ידי המנחה ויוחזרו לך בצירוף הערות המתייחסות לתשובות.

לכל מטלה נקבע משקל. יש לצבור 36 נקודות. חובה להגיש את כל המטלות.

ללא צבירת 36 נקודות בהגשת מטלות לא ניתן יהיה לגשת לבחינת הגמר

לתשומת לבכם!
ציון סופי מחושב רק לסטודנטים שעברו את בחינת הגמר בציון 60 ומעלה והגישו את כל המטלות בציון 60 לפחות.

כל סטודנט יכין את הממ"נים לבדו. אין להגיש את הממ"נים בזוגות (או קבוצות) !

3. התנאים לקבלת נקודות זכות

א. הגשת מטלות במשקל כולל של **36** נקודות לפחות עם ציון מינימלי של **60** נקודות בכל אחת מהמטלות שהוגשו.
ב. ציון של לפחות **60** נקודות בבחינת הגמר.

4. הדרכה לפתרון תרגילי התכנות

תרגילי התכנות בקורס זה דורשים מאמץ ניכר. התרגילים לכשעצמם אינם קשים באופן מיוחד אולם הם דורשים הכרה והבנה טובה של החומר המוצע כחומר רקע (ראו סעיף "חומר קרע" בגוף כל ממ"ן)

למרות שהקוד הנדרש בסופו של דבר בתרגילי התכנות איננו ארוך, סביר להניח כי תקדישו לתרגילים שעות רבות. תכנות מערכת הפעלה, דורש ניסיון, ולמרבה העצב רכישת הניסיון כרוכה לרוב גם בהקדשת זמן. עם זאת, התרגילים תוכננו כך שיעסקו מעט ככל האפשר בנושאים שמטבעם הם טכניים בלבד.

בפתרון התרגילים אנו מציעים את השלבים הבאים:

א. קראו היטב את דרישות התרגיל והבהירו לעצמכם מה הבעיות שעלולות להתעורר בעת יישומו.

ב. קראו את החומר המוצע כחומר רקע (ראו סעיף "חומר קרע" בגוף כל ממ"ן). לצורך זה מצויים
בידכם ארבעה מקורות, עיינו בהם על פי הסדר הבא:

1. ספר הקורס , Modern Operating Systems, המספק את הרקע התיאורטי.
2. המדריך למתכנת המערכת, The GNU C library reference manual, מתאר את פעולת קריאות המערכת ברוב מערכות UNIX הקיימות
3. הפקודה "man *command-name*" ב-UNIX מאפשרת לקבל מידע על פקודות, פונקציות ספריה, וקריאות מערכת, כפי שהן ממומשות במערכת שבידך.
4. מידע נוסף שמכיל דוגמאות קוד והסברים אפשר למצוא באינטרנט, בפרט באתרים שכתובותיהם מצויים בקטגוריה "אתרים ברשת" (ראו את הדף הראשי של אתר הקורס).

ג. בעת כתיבת הקוד, הקפידו על הכללים המקובלים, בהנדסת תוכנה. רוב הדרישות המפורטות כאן מוכרות לכם בודאי מקורסים קודמים אומנם ישנן דרישות ייחודיות לקורס במערכות הפעלה. לקיום הדרישות הללו קיימת השפעה על ציון הממ"ן:

1. מתן שמות משמעותיים למשתנים.
2. הימנעות משימוש במספרים שרירותיים.
3. כתיבת פונקציות קצרות.

4. תיעוד סביר. הכוונה לתיעוד מתומצת של פעולות התוכנית, של פונקציות ושל משתנים. כמו כן, יש לרשום בתחילת כל קובץ קוד שמוגש את הפרטים האישיים (שם מלא ומספר סטודנט) ותיאור קצר של תוכן הקובץ.
5. יש להקפיד על שימוש בשמות המוגדרים במטלה.
6. אין להשתמש ב goto. ליציאה מלולאות ניתן להשתמש במידת הצורך ב continue או break.
7. מבנה מדורג. מודולים ופונקציות קצרות וללא אפקטים משניים.
8. Indentation.
9. משפטי תנאי קצרים.
10. כל יציאה בגלל שגיאה חייבת להיות מתועדת. למשל, באמצעות הפונקציה perror().
11. בכל מקרה יש לבדוק את הערך המוחזר על ידי קריאות מערכת.
12. בכל מקרה יש לבדוק את נכונות הקלט.
13. התוכנית לא תיפול עקב שגיאה/תקלה כלשהי. במידה וקורה אירוע בלתי צפוי, על התוכנית להודיע על כך ולסיים את עבודתה.
14. אין להשתמש בפונקציה system().
15. יש לשחרר את כל המשאבים שאינם בשימוש.
16. הוראות קומפילציה יש לכתוב בשפת ההוראות של תוכנית השירות make ולהגישם בקובץ בשם makefile.
17. חובה להשתמש בדגל (flag) "-Wall" בזמן קומפילצית התוכניות

בונוס

במקרים יוצאי דופן, כאשר מוגשת תוכנית טובה במיוחד או כזו שעושה למעלה ממה שנדרש, תישקל האפשרות להוסיף עד 5 נקודות בונוס. בכל מקרה שהנכם מתכוונים להגיש תוכנית מעין זו, שימו לב כי:

1. כל הדרישות מהתוכנית המקורית יתקיימו.
2. כל תוספת תהיה מתועדת היטב.
3. תוספות המכילות שגיאות עלולות להוריד מהניקוד הסופי גם אם התוספות לא נדרשו במטלה. כוונות טובות אינן מובילות בהכרח לתוצאה הרצויה.

מטלת מנחה (ממ"ן) 11

הקורס: "מערכות הפעלה"

חומר הלימוד למטלה: ראו פירוט בסעיף "רקע"

מספר השאלות: 6 **משקל המטלה: 12**

סמסטר: 2016א **מועד אחרון להגשה:** 26.11.2015

הגשת המטלה: שליחה באמצעות מערכת המטלות המקוונת באתר הבית של הקורס.
הסבר מפורט ב"נוהל הגשת מטלות המנחה".

החלק המעשי (70%)

כללי

בממ"ן זה עליכם לממש שתי ספריות לעבודה עם תהליכונים (threads) ברמת המשתמש (user-level). אחת הספריות תממש סמפורים בינאריים לעבודה עם קטעים קריטיים וספריה שנייה תממש מספר פונקציות המאפשרות יצירה והרצה של תהליכונים ברמת המשתמש ומדידת זמן הריצה ל profiling של תוכניות המשתמשות בספרייה זו.

מטרה

- הכרת ההיבטים המעשיים של מימוש תהליכונים ברמת המשתמש
- שימוש בסיגנלים
- שימוש ב-non-local branching
- timers
- profiling
- קטעים קריטיים

רקע

א) פרקים 2.3.5, 2.5.1, 2.2.1, 2.2.2, 2.2.3 בספר של Tanenbaum, "Modern operating systems".

ב) פרק 24.3 של The GNU C library

ג) פרק 23.4 של The GNU C library

ד) פרק "Libraries" מחוברת "Ubuntu 12.04 programming environment, making first steps"

ה) man pages של Linux - מידע על קריאות מערכת ופונקציות הבאות:

alarm, sigfillset, sigaction, swapcontext, getcontext, makecontext, steitimer, kill,getpid

תיאור המשימה

בממ"ן זה עליכם לממש שתי ספריות סטטיות:

1) libut.a - ספרייה פשוטה לעבודה עם תהליכונים ברמת המשתמש, שה-API שלה מוגדר בקובץ ut.h. קובץ זה מכיל תיאור מפורט לגבי תפקידה של כל פונקציה שעליכם לממש (אין לשנות קובץ

זה,אך כמובן שבמידת הצורך ניתן להגדיר פונקציות עזר בקובץ C). הספרייה תתמוך רק בפעולות הבסיסיות ביותר, שהן יצירת התהליכונים, הרצתן ותזמונן. על מנת שלא להפוך את המשימה למסובכת מדי, הספרייה תממש רק מודל פשוט של שימוש בתהליכונים המבוסס על ההנחות הבאות:

א. כל תהליכון מריץ פונקציה אינסופית שמקבלת פרמטר יחיד מטיפוס int ומחזירה void.
לא נטפל בסיום תהליכונים ובבדיקת סטטוס היציאה.

ב. אין הוספה דינאמית של תהליכונים. המשתמש קודם יצור את כל התהליכונים, וא"כ יקרא ל-()ut_start כדי להריץ את כל התהליכונים.

ג. כל התהליכונים הם בעלי אותה עדיפות. תזמון התהליכונים יהיה בשיטת round-robin, כאשר גודל ה-quantum הוא שנייה אחת.

ד. שימו לב שלא הגדרנו מצב blocked לתהליכונים. זאת מפני שבמודל שלנו ההנחה היא שתהליכונים לא מבצעים פעולות הגורמות לחסימה (blocking calls). לאחר הביצוע של ()ut_start, כל תהליכון יכול להיות באחד משני המצבים - רץ או מוכן לריצה. וודאו שאתם מבינים כי בהנחה כזאת כלל לא נצטרך לשמור את מצב ההליכונים מכוון שמנגנון התזמון שלנו תמיד יבחר את התהליכון הבא בתור ויריץ אותו.

בשלב ראשון של הכנת הממ"ן קראו את הסעיפים א), ב) וג) מחומר רקע והריצו והבינו את התוכניות demo1.c, demo2.c, demo3.c שסיפקנו לכם. התוכנית הראשונה מדגמה כיצד מתאפשר לשים "שעון מעורר" לתהליך ב Linux. התוכנית השנייה מרחיבה את הראשונה ומדגימה כיצד אפשר ליצור 2 ניבים של ריצה בתוכנית באמצעות המנגנון המכונה non-local jumping. התוכנית השלישית מדגימה כיצד אפשר לבצע רישום של זמן ריצה של תוכנית לצורך ה profiling.

בשלב שני עליכם לממש את הממשק המוגדר הקובץ ut.h. הממשק מגדיר פונקציות לאתחול הספריה, ליצירת תהליכון חדש ולהרצת התהליכונים שנוצרו. ut.h מממשת את מודל התהליכונים הפשוט שתיארנו לעיל. שימו לב ש demo2.c מדגימה כיצד ליצור 2 תהליכונים. אתם מתבקשים להכליל את הפתרון למספר תהליכונים. לכן, לאחר שהשלמתם את שני השלבים הקודמים כל שנותר לעשות הוא להעביר חלקים של הקוד מ demo2.c ל ut.c עם שינויים מינוריים.

ב ut.h עליכם לממש את `ut_get_vtime` המשמשת למדידת זמן הריצה של תהליכון. השתמשו בקוד של demo3.c שמשתמשת בשעון מסוג ITIMER_VIRTUAL שישלח סיגנל `SIGVTALRM` כל 100msec (10 פעמים לשנייה). בכל פעם שהסיגנל מתקבל,יש להוסיף 100msec לשדה הזמן הווירטואלי של התהליכון האקטיבי בזמן קבלת הסיגנל.

2) libbinsem.a - ספרייה של סמפורים בינאריים שנועדו לשימוש ע"י התהליכונים מהסעיף הראשון. הקובץ binsem.h מגדיר את הטיפוס של סמפור בינארי ומתאר את הפונקציות הרלוונטיות (אין לשנות קובץ זה). עליכם לממש את הפונקציות שמוצהרות בקובץ זה, תוך כדי שימוש במקרו ()xchg המוגדר בקובץ atomic.h. כמו כן, תסתמכו על העובדה שהחלפת התהליכונים מתבצעת כתוצאה מקבלת הסיגנל SIGALRM כדי לממש את ההמתנה ב-()binsem_down (כפי שפורט בסעיף הקודם, לתהליכונים שעליכם לממש לא מוגדר מצב

blocked. יש לדמות את המצב ע״י כך שתהליכון ה״ממתין״ בסמפור מייד לאחר קבלת ה-CPU ישלח סיגנל SIGALRM שיגרום להפעלת המתזמן ומעבר לתהליכון הבא).

לצורך הבדיקה של שתי הספריות סיפקנו לכם פתרון של בעיית הפילוסופים הסועדים בקובץ ph.c. בעיית הפילוסופים הסועדים מתוארת בפרק 2.5.1 בספר של Tanenbaum. כל פילוסוף רץ כתהליכון נפרד (לצורך זה משתמשים בספריית התהליכונים שהממשק שלה הוגדר ב ut.h. התהליכונים משתמשים בסמפורים שהוגדרו ב binsem.h). התוכנית תופעל ע״י הפקודה ״ph <N>״, כאשר N (בטווח מ-2 עד 32) הוא מספר התהליכונים (פילוסופים). התוכנית תופסק ע״י הקשת ״Ctrl-C״, לפני היציאה יודפסו זמני השימוש ב-CPU של כ״א מהתהליכונים.

כדי לקמפל את תוכנית הפילוסופים עם הספריות שתכתבו, תשתמשו ב Makefile שסיפקנו. שימו לב שעליכם לשנות את ה Makefile לפני ההגשה (ראו סעיף ״הגשה״ בהמשך).

טיפול בשגיאות

יש תמיד לבדוק את ערכי החזרה של קריאות מערכת ופונקציות סטנדרטיות של C. במקרה של כשלון, יש לפעול כפי שמוגדר בקבצים ut.h ו-binsem.h. בנוסף, במקרה של כשלון המערכת תוך כדי ביצוע של signal handler(s) בספריית התהליכונים, יש להודיע על השגיאה באמצעות perror() ולהפסיק את הביצוע ע״י exit(1).

הגשה

יש להגיש **<u>כל</u>** קבצי הקוד Makefile המייצר שתי ספריות סטטיות: libut.a ו-libbinsem.a
אין להגיש קבצים מקומפלים. ראה הוראות הגשה כלליות בחוברת הקורס.

את הקבצים המוגשים יש לשים בקובץ ארכיון בשם exYZ.zip (כאשר YZ הנו מספר המטלה). הכנת קובץ ארכיון מתבצעת ע״י הרצת הפקודה הבאה משורת הפקודה של Ubuntu:

zip exYZ.zip <ExYZ files>

<u>הערה חשובה: בכל קובץ קוד שאתם מגישים יש לכלול כותרת הכוללת תיאור הקובץ, שם הסטודנט ומספר ת.ז.</u>

פתרון ביה״ס

קיבלתם את שתי הספריות, libut.a ו- libbinsem.a , כפי שמומשו על ידינו. תוכלו להיעזר בהן בהכנת הממ״ן/ למשל/ לקמפל את תוכנית הבדיקה ph עם ספרייה אחת משלכם (שאותה אתם רוצים לבדוק) וספרייה השנייה של פתרון ביה״ס.

<u>הערה: תוך כדי העבודה על הממ״ן תצטרכו להכיר ולהבין מספר נושאים שאינם פשוטים - זהו הקושי של ממ״ן זה. יחד עם זאת, הממ״ן לא ידרוש מכם הרבה עבודת תכנות. ניתן לממש את שתי הספריות בכ-100 שורות קוד בסה״כ.</u>

החלק העיוני (30%)

שאלה 2 (5%)

הסבירו מהם 2 התפקידים העיקריים של מערכת הפעלה ומה חשיבותם.

שאלה 3 (10%)

א) מהי פעולת ה TRAP (TRAP instruction). תארו מתי היא מתבצעת ומה קורא בעת ביצועה.

ב) הסבירו מה קורה בעת הקריאה לפונקציית write של ה C library. בפרט הסבירו כיצד עוברים הפרמטרים של ה write למערכת הפעלה Linux וכיצד המערכת מטפלת ב write.

ג) מה ההבדל בין write ל printf? תוכלו להעזר בקבצי מקור של C library מ http://www.gnu.org/software/libc/

שאלה 4 (5%)

הסבר את מדוע פתרון התור (strict alternation), איננו מהווה פתרון סביר. איזה תנאי/ים הוא מפר.

שאלה 5 (5%)

האם מדיניות הוצאת תהליכונים מתור המתנה של סמפור יכולה להיות שונה מ first in first out? אם כן, הבר מדוע. אם לאו, תאר את הבעיה.

שאלה 6 (5%)

הוכיחו כי בפתרון של Peterson תהליכים אינם ממתינים זמן אינסופי על מנת להיכנס לקטע קריטי. בפרט הוכיחו כי תהליך שרוצה להיכנס לקטע קריטי לא ממתין יותר ממה שלוקח מתהליך אחר להיכנס ולעזוב את הקטע הקריטי.

הגשת החלק העיוני

החלק העיוני יוגש כקובץ Word או כקובץ pdf. שם הקובץ צריך להיות exYZ.pdf או exYZ.doc (כאשר YZ הנו מספר המטלה).

מטלת מנחה (ממ"ן) 12

הקורס: "מערכות הפעלה"

חומר הלימוד למטלה: ראו פירוט בסעיף "רקע"

מספר השאלות: 5 **משקל המטלה: 12**

סמסטר: 2016א **מועד אחרון להגשה:** 24.12.2015

הגשת המטלה: שליחה באמצעות מערכת המטלות המקוונת באתר הבית של הקורס.
הסבר מפורט ב"נוהל הגשת מטלות המנחה".

החלק המעשי (80%)

בממ״ן זה עליכם לכתוב ספריה להקצאה דינמית של זיכרון התומכת בריבוי תהליכונים. את הממשקים שיש לממש, ניתן לראות בקובץ mtmm.h.

מטרה

- ניהול הזיכרון
- ריבוי תהליכונים

רקע

א) הסבר ליצירת ספריות סטטיות בפרק "Libraries" מתוך החוברת Ubuntu 12.04 programming environment, Making first steps

ב) פרק 8.7, ספר מאת "C Programming Language" Ritchie & Kernighan

ג) הסבר לניהול זיכרון בסביבה מרובת תהליכונים בתוך "Implementation of multithread memory management"

תיאור המשימה

תכנות מקבילי בשפות C ו C++ בתוכניות כגון שרתי אינטרנט, מנהלי מסדי נתונים, שרתי חדשות, ויישומים מדעיים הופכים הקצאה ושחרור של זיכרון לצוואר בקבוק שמגביל ביצועי מערכת ומגביל יכולת ניצול של מספר גדול של מעבדים במערכות שהן מרובות מעבדים.

בממ"ן זה נממש ספרייה להקצאה דינאמית של זיכרון שתנצל בצורה טובה את קיום ריבוי המעבדים. קראו את חומר הרקע בסדר המופיע לעיל. בפרק 8.7 של K&R תראו פתרון פשוט לבעית הקצאה ושחרור של זיכרון המשתמש בערמה (heap) אחד. בהסבר המופיע ב Implementation of multithread memory management תואר כיצד לממש את ספרייה להקצאת זיכרון בצורה המתאימה לריבוי תהליכונים. במשפט אחד אנחנו נשתמש במספר ערמות – אחת לכל תהליכון ועוד ערמה נוספת עבור כל התהליך כולו.

טיפול בשגיאות

יש לבדוק את ערכי החזרה של קריאות מערכת ופונקציות סטנדרטיות של C. במקרה של כשלון, יש לפעול כפי שמוגדר בקובץ mtmm.h.

הגשה

יש להגיש **כל** קבצי הקוד ו Makefile המייצר ספריה סטטית libmtmm.a. את הקבצים המוגשים יש לשים בקובץ ארכיון בשם exYZ.zip (כאשר YZ הנו מספר המטלה). הכנת קובץ ארכיון מתבצעת ע"י הרצת הפקודה הבאה משורת הפקודה של Ubuntu:

<zip exYZ.zip <ExYZ files

<u>הערה חשובה: בכל קובץ קוד שאתם מגישים יש לכלול כותרת הכוללת תיאור הקובץ, שם הסטודנט ומספר ת.ז.</u>

<u>פתרון ביה"ס</u>

קיבלתם ספריה סטטית libmtmmSSol.a , כפי שמומשו על ידינו. תוכלו להיעזר בה בהכנת הממ"ן,למשל-לקמפל איתה את תוכניות הבדיקה שלנו וכמובן עם תוכניות בדיקה שלכם.

החלק עיוני (20%)

שאלה 1 – (5%)

א. איזה סוג של מידע נדרש לאלגוריתם הבנקאי?

ב. האם אלגוריתם הבנקאי מונע את היווצרות ה deadlock?

ג. תארו אסטרטגיה מעשית להימנעות מ deadlocks אשר "תוקפת" את התנאי של המתנה מעגלית.

שאלה 2 – (5%)
כיצד משפיע גודל דף על גודל ה working set?

שאלה 3 – (5%)

א. כיצד ניתן לזהות מצב של סחרור (thrashing)?

ב. כיצד ניתן להתגבר על מצב סחרור?

שאלה 4 – (5%)

טבלת הדפים של תהליך במערכת עם זיכרון וירטואלי נראית כך. כל המספרים הם דצימליים, מתחילים מאפס, וכל הכתובות הן כתובות של בייט בזיכרון. גודל הדף הוא 1024 בייטים.

Page Number	Valid bit	Frame Number
0	1	4
1	1	7
2	0	-
3	1	2
4	0	-
5	1	0

לאילו כתובות פיזיות, אם יש כאלו, ימופו הכתובות הוירטואליות הבאות: 1052, 2221, 5499.

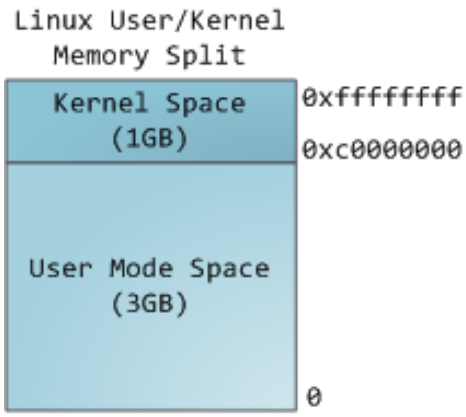
הגשת החלק העיוני

החלק העיוני יוגש כקובץ Word או כקובץ pdf. שם הקובץ צריך להיות exYZ.pdf או exYZ.doc (כאשר YZ הנו מספר המטלה).

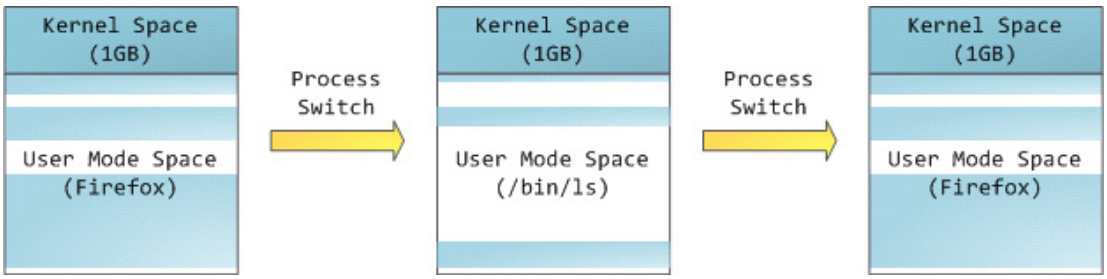
Task 1 - Understanding the virtual memory address space and basic dynamic space allocation

Task objective: The objective of this task is to understand the working concept of virtual address space management. The material presented in this task is a revised adaptation of [1].

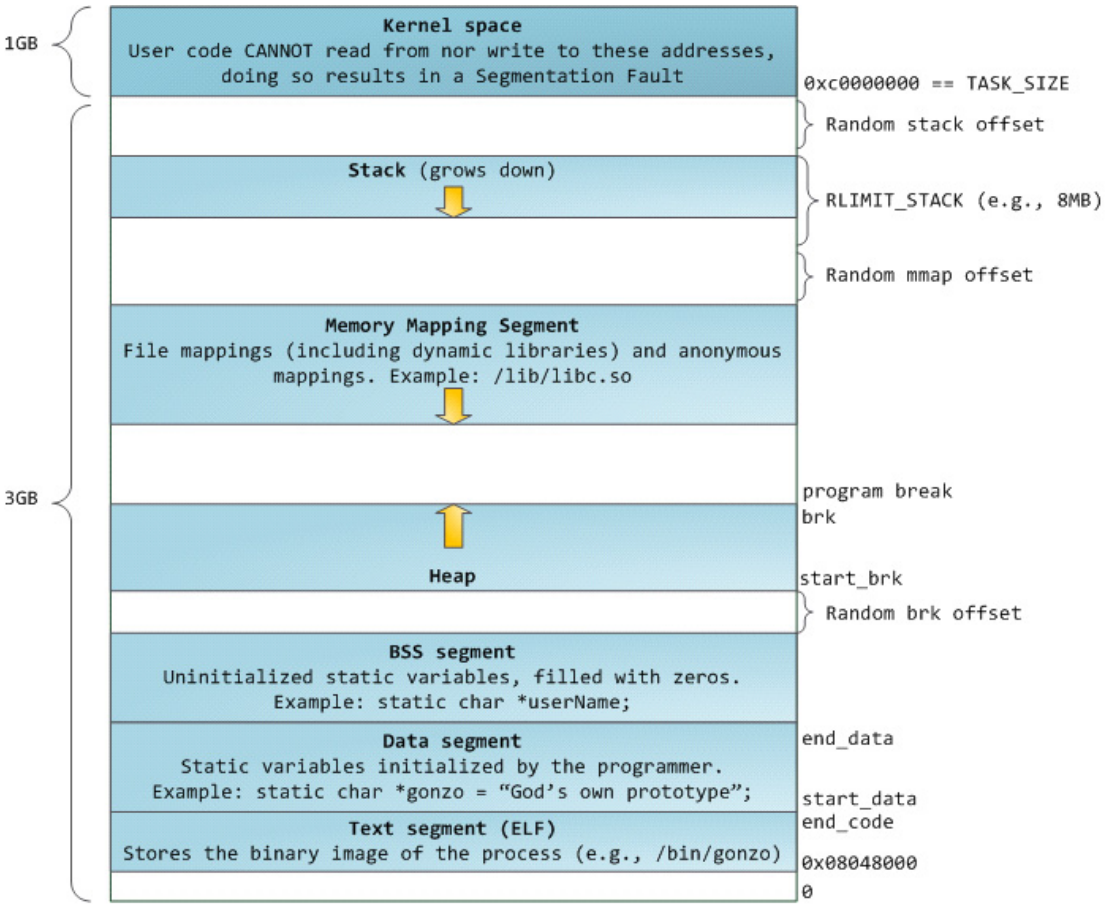
Task description: On 32 bit machines the size of a process virtual address space in Linux is 4Gb. This space is split 1Gb for kernel space and 3Gb for user mode.



This does **not** mean the kernel uses that much physical memory, only that it has that portion of address space available to map whatever physical memory it wishes. Kernel space is flagged in the page tables as exclusive to privileged mode, hence a page fault is triggered if user-mode programs try to touch it. In Linux, kernel space is constantly present and maps the same physical memory in all processes. Kernel code and data are always addressable, ready to handle interrupts or system calls at any time. By contrast, the mapping for the user-mode portion of the address space changes whenever a process switch happens:



Blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. In the example above, Firefox has used far more of its virtual address space due to its legendary memory hunger. The distinct bands in the address space correspond to **memory segments** like the heap, stack, and so on. Here is the standard segment layout in a Linux process:



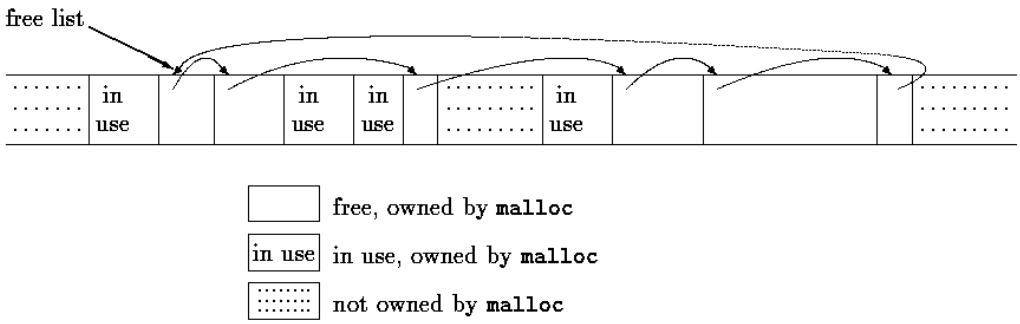
The topmost segment in the process address space is the stack, which stores local variables and function parameters in most programming languages. Calling a method or function pushes a new **stack frame** onto the stack. The stack frame is destroyed when the function returns. This simple design, possible because the data obeys strict LIFO order, means that no complex data structure is needed to track stack contents – a simple pointer to the top of the stack will do. Pushing and popping are thus very fast and deterministic. Also, the constant reuse of stack regions tends to keep active stack memory in the cpu caches, speeding up access. Each thread in a process gets its own stack.

Below the stack, we have the memory mapping segment. Here the kernel maps contents of files directly to memory. Any application can ask for such a mapping via the Linux mmap() system call. Memory mapping is a convenient and high-performance way to do file I/O, so it is used for loading dynamic libraries. It is also possible to create an **anonymous memory mapping** that does not correspond to any files, being used instead for program data. In Linux, if you request a large block of memory via malloc(), the C library will create such an anonymous mapping instead of using heap memory. 'Large' means larger than `MMAP_THRESHOLD` bytes, 128 kB by default and adjustable via mallopt().

Speaking of the heap, it comes next in our plunge into address space. The heap provides runtime memory allocation, like the stack. Bur heap is meant for data that

must outlive the function doing the allocation, unlike the stack. Most languages provide heap management to programs. Satisfying memory requests is a joint affair between the language runtime and the kernel. In C, the interface to heap allocation is malloc() and friends.

If there is enough space in the heap to satisfy a memory request, it can be handled by the language runtime without kernel involvement. Otherwise the heap is enlarged. Many heap management schemes are available. One simple heap management scheme is explained in chapter 8.7 of [2]. This scheme enlarges heap using the brk() system call to make room for the requested block. **Read chapter 8.7 to see a this implementation of the heap management that is a linked list of free and used chunks of memory (aka heap segment).**

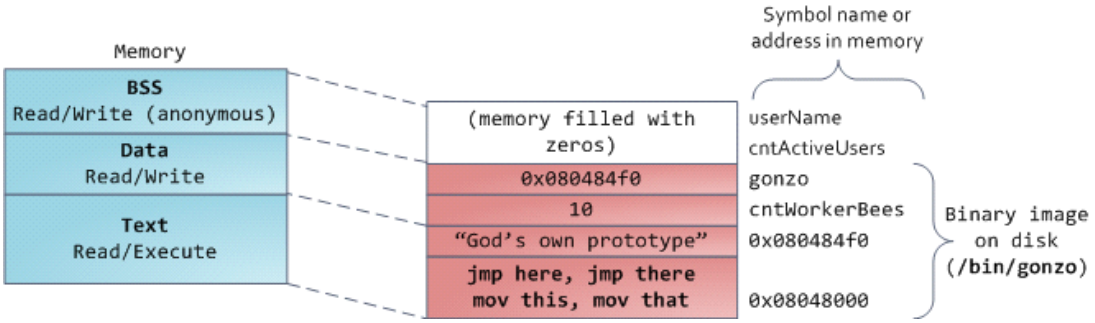


Finally, we get to the lowest segments of memory: BSS, data, and program text. Both BSS and data store contents for static (global) variables in C. The difference is that BSS stores the contents of *uninitialized* static variables, whose values are not set by the programmer in source code. The BSS memory area is anonymous: it does not map any file (executable file or swap file). If you say `static int cntActiveUsers`, the contents of `cntActiveUsers` live in the BSS.

The data segment, on the other hand, holds the contents for static variables initialized in source code. This memory area **is not anonymous**. It maps the part of the program's binary image that contains the initial static values given in source code. So if you say `static int cntWorkerBees = 10`, the contents of cntWorkerBees live in the data segment and start out as 10. Even though the data segment maps a file, it is a **private memory mapping**, which means that updates to memory are not reflected in the underlying file. This must be the case, otherwise assignments to global variables would change your on-disk binary image.

The data example in the diagram is trickier because it uses a pointer. In that case, the *contents* of pointer `gonzo` – a 4-byte memory address – live in the data segment. The actual string it points to does not, however. The string lives in the **text** segment, which is read-only and stores all of your code in addition to tidbits like string literals. The text segment also maps your binary file in memory, but writes to this area earn your program a Segmentation Fault. This helps prevent pointer bugs, though not as

effectively as avoiding C in the first place. Here's a diagram showing these segments and our example variables:



You can examine the memory areas in a Linux process by reading the file `/proc/pid_of_process/maps`. For example, run "cat /proc/self/maps" to see the memory map of cat when is runs on the system.

```
[redacted] ~ # cat /proc/self/maps
08048000-08051000 r-xp 00000000 00:0c 301        /bin/cat
08051000-08052000 r--p 00008000 00:0c 301        /bin/cat
08052000-08053000 rw-p 00009000 00:0c 301        /bin/cat
08053000-08074000 rw-p 00000000 00:00 0          [heap]
b722a000-b73bc000 r--p 0027c000 00:0c 17690      /usr/lib/locale/locale-archive
b73bc000-b75bc000 r--p 00000000 00:0c 17690      /usr/lib/locale/locale-archive
b75bc000-b75bd000 rw-p 00000000 00:00 0
b75bd000-b7711000 r-xp 00000000 00:0c 3323       /lib/libc-2.12.2.so
b7711000-b7713000 r--p 00154000 00:0c 3323       /lib/libc-2.12.2.so
b7713000-b7714000 rw-p 00156000 00:0c 3323       /lib/libc-2.12.2.so
b7714000-b7717000 rw-p 00000000 00:00 0
b771c000-b771d000 rw-p 00000000 00:00 0
b771d000-b7739000 r-xp 00000000 00:0c 3432       /lib/ld-2.12.2.so
b7739000-b773a000 r-xp 00000000 00:00 0          [vdso]
b773a000-b773b000 r--p 0001c000 00:0c 3432       /lib/ld-2.12.2.so
b773b000-b773c000 rw-p 0001d000 00:0c 3432       /lib/ld-2.12.2.so
bfa9f000-bfac0000 rw-p 00000000 00:00 0          [stack]
```

We can see from this figure that segment may contain many areas. For example, each memory mapped file normally has its own area in the mmap segment, and dynamic libraries have extra areas similar to BSS and data. As you can see the picture is quite complex. Also, sometimes the vocabulary might differ in several ways. For example, people may say "data segment" meaning all of data + bss + heap.

Task 2 – Understanding how are we going to implement dynamic memory allocation

Task objective: We are going to implement a library for dynamic memory allocation. Previous task got you acquainted with a simple dynamic allocator using the brk system call and a linked list of free regions. The objective of this task is to understand the description of the dynamic memory allocation scheme we are going to implement

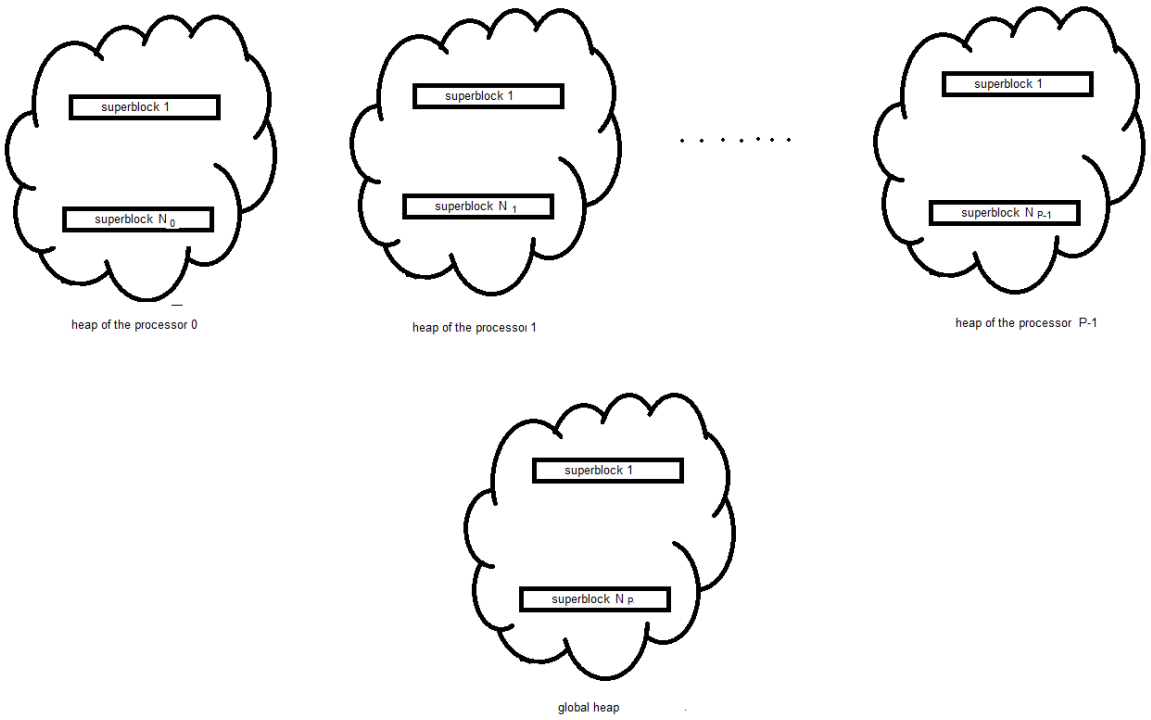
as a part of our assignment (maman 12) with is different from the brk + linked list scheme.

Task description:

The implementation of memory management depends greatly upon operating system and architecture and even for the same operating system and architecture several memory allocators may exist [3]. In our assignment we will implement a simple version of a Hoard memory allocator [4].

Our version of the allocator will have P + 1 heaps where P is the number of processors of the system. **According to Hoard, heap is a "mmap"-ed area in the virtual address space. That means, Hoard's definition of the heap is different from the brk+linked list scheme.** The idea of having P heaps is to allow dynamic memory allocations management for each processor on a dedicated heap. The idea of having one more heap (which is also called a global heap) is to allow underutilized chunks of memory from per-processor heaps to be moved from these per-processor heaps to the global one. We will define the underutilized chunk propertied later.

Each heap "owns" a number of so called superblocks or in putting differently, each heap (both global and per-processor) is a collection of superblocks.



When there is no memory available in any superblock on a thread's heap, the allocator obtains a superblock from the global heap if one is available. If the global heap is also empty, Hoard creates a new superblock by requesting virtual memory from the operating system and adds it to the thread's heap. Hoard does not return empty superblocks to the operating system. It instead makes these superblocks available for reuse.

Each superblock is an array of some number of blocks (objects) and contains a free list of its available blocks maintained in LIFO order to improve locality. All superblocks are of the same size (S). The superblock size S is defined in *mtmm.h* as SUPERBLOCK_SIZE. Objects larger than half the size of a superblock are managed directly using the virtual memory system (i.e., they are allocated via mmap and freed using munmap). All of the blocks in a superblock are in the same size class.

From here on, we assume a single size class in the discussion below for clarity of explanation. One can easily generalize the discussion to multiple size classes.

Memory allocator will maintain *usage statistics* for each of P+1 heaps. These statistics are:

u(i) - the amount of memory in use ("live") in heap *i*

and

a(i) - the amount of memory held in heap *i* that was allocated by the memory allocator from the operating system.

Hoard moves superblocks from a per-processor heap to the global heap when the per-processor heap crosses the emptiness threshold: more than ***f***, the empty fraction, of its blocks are not in use (u(i) < (1 − ***f)a(i)), and there are more than some number K of superblocks' worth of free memory on the heap (u(i) < a(i) − K ∗ S).*** The empty fraction f is defined in mtmm.h as HOARD_EMPTY_FRACTION and K is defined as HOARD_K.

As long as a heap is not more than f empty, and has K or fewer superblocks, Hoard will not move superblocks from a per-processor heap to the global heap. Whenever a per-processor heap does cross the emptiness threshold, Hoard transfers one of its superblocks that is at least f empty to the global heap. Always removing such a superblock whenever we cross the emptiness threshold maintains the following invariant on the per-processor heaps: (u(i) ≥ a(i) − K ∗ S) ∨ (u(i) ≥ (1 − f)a(i)). When we remove a superblock, we reduce u(i) by at most (1 − f)S but reduce a(i) by S, thus restoring the invariant.

Let's have a look at the example of how the allocator manages superblocks? For simplicity, we assume there are two threads and heaps (thread i maps to heap i). In this example (which reads from top left to top right, then bottom left to bottom right), the empty fraction f is 1/4 and K is 0. Thread 1 executes code written on the left-hand side of each diagram (prefixed by "t1:") and thread 2 executes code on the right-hand side (prefixed by "t2:"). Initially, the global heap is empty, heap 1 has two superblocks (one partially full, one empty), and heap 2 has a completely-full superblock. The top left diagram shows the heaps after thread 1 allocates x9 from heap 1. Hoard selects the fullest superblock in heap 1 for allocation. Next, in the top right diagram, thread 1 frees y4, which is in a superblock that heap 2 owns. Because heap 2 is still more than 1/4 full, Hoard does not remove a superblock from it. In the bottom left diagram, thread 2 frees x2, which is in a superblock owned by heap 1. This free does not cause heap 1 to cross the emptiness threshold, but the next free (of x9) does. Hoard then moves the completely-free superblock from heap 1 to the global heap.

Finally let's present the pseudocode of the malloc and free. ***NOTE, that this time size classes are also referred.***

malloc(*sz*)

1. If *sz* > *S/2*, allocate the superblock from the OS and return it.
2. *i* ← hash(current thread)
3. **Lock heap *i* .** ***(Note: actually you should lock appropriate size class in heap i.)***
4. Scan heap *i*'s list of superblocks from full to least (for the size class corresponding to *sz*)
5. If there is no superblock with free space {
6. Check heap 0 (the global heap) for a superblock. ***(That means that here one should lock an appropriate size class in heap 0)***
7. If there is none
8. Allocate *S* bytes as superblock *s* and set owner to heap *i* ***(at appropriate size class)***
9. Else
10. Transfer the superblock *s* to heap *i* ***(at appropriate size class)***
11. *u0* ← *u0* – *s.u;* ***(at appropriate size class)***
12. *ui* ←*ui* + *s.u* ***(at appropriate size class)***
13. *a0* ← *a0* - *S;* ***(at appropriate size class)***
14. *ai* ← *ai* + *S* ***(at appropriate size class)***
15. *ui*←*ui* + *sz;* ***(Actually one should account a size of sz + sizeof (chunk header))***
16. *s.u* ← *s.u* + *sz* ***(Actually one should account a size of sz + sizeof (chunk header))***
17. **Unlock heap *i*** ***(Note: actually you should lock appropriate size class in heap i.)***
18. Return a block from the superblock

free(ptr)

1. If the block is "large"
2. Free superblock to OS and return
3. Find the superblock *s* this blocks comes from and **lock it**
4. **Lock heap *i*, the superblock's owner;** ***(Note: actually you should lock appropriate size class in heap i)***
5. Deallocate the block from the superblock
6. *ui* ←*ui* – block size
7. *s.u* ← *s.u* – block size
8. If (*i* = 0) **unlock heap *i*, superblock *s*** and return
9. If (*ui* < *ai* – *K***S*) and (*ui* <*(1-f)***ai*)
10. Transfer a mostly-empty superblock *s1* to heap 0 (the global heap) ***(at appropriate size class)***
11. u0 ←u0 + s1.u; ui ← ui – s1.u ***(at appropriate size class)***
12. a0 ← a0 + S; ai ← ai – S ***(at appropriate size class)***
13. **Unlock heap *i* and superblock *s***

Explanation of the malloc

Hoard directly allocates "large" objects (size > S/2) via the virtual memory system. When a thread on processor i calls malloc for small objects, Hoard locks heap i and gets a block of a superblock with free space, if there is one on that heap (line 4). If there is not, Hoard checks the global heap (heap 0) for a superblock. If there is one, Hoard transfers it to heap i, adding **the number of bytes in use in the superblock s.u** to u(i), and the total number of bytes in the superblock S to a(i) (lines 10–14). If there are no superblocks in either heap i or heap 0, Hoard allocates a new superblock and inserts it into heap i (line 8). Hoard then chooses a single block from a superblock with free space, marks it as allocated, and returns a pointer to that block.

Note: the definition of s.u is the number of bytes in use in the superblock s.
Explanation of the free
Each superblock has an "owner" (the processor whose heap it's in). When a processor frees a block, Hoard finds its superblock (through a pointer in the block's header). (If this block is "large", Hoard immediately frees the superblock to the operating system.) It first locks the superblock and then locks the owner's heap. Hoard then returns the block to the superblock and decrements u (i). If the heap is too empty (u(i) < a(i) − K ∗ S and u(i) < (1 − f)a(i)), Hoard transfers a superblock that is at least f empty to the global heap (lines 10-12). Finally, Hoard unlocks heap i and the superblock.

Task 3 – Run the school solution and compare with regular malloc

We have provided a *libmtmmSSol.a* shared library that re-implements the regular malloc, free and realloc supplied by glibc and defined in stdlib.h. We also have provided a driver called *linux-scalability.c* that creates multiple threads each allocating/freeing memory chunks. Makefile that we have supplied compiles to *linux-scalability* executable file.

You can choose one of the following options:

1) uncomment the line *MYFLAGS = -g -O0 -Wall -fno-builtin-malloc -fno-builtin-calloc -fno-builtin-realloc -fno-builtin-free* in our Makefile to link to *linux-scalability* with hoard memory allocator (and comment the line *MYFLAGS = -g -O0 -Wall*)
2) uncomment the line *#MYFLAGS = -g -O0 –Wall* in our Makefile to link to *linux-scalability* with a standard memory allocator (and comment other " *MYLIBS =XYZ"* lines)

After the *Makefile* modification run *"make clean; make"* and launch the driver. E.g. by

./linux-scalability 512 10000 10

The paramaters of the driver deserve a short explanation. 512 is the size of chunks allocated via malloc. 10000 is the number of malloc calls and 10 is the number of threads each one performing 10000 mallocs of 512 byte.

Recompile the *linux-scalability* with different memory allocation library and compare results. The following table demonstrates the comparison of results obtained with the hoard memory allocator and the standard memory allocator:

	./linux-scalability 512 1000000 50	
	Hoard memory allocator	**Standard memory allocator**

	Average execution time	standard deviation	Average execution time	standard deviation
	6.831516	2.918605	2.474888	0.344976
	6.51319	2.799645	2.471997	0.447987
	5.6687	2.440067	2.671545	0.457854
	2.900224	1.20315	5.700411	2.652924
	10.472085	2.947556	6.641511	2.966338
	5.595637	2.60138	4.995677	2.00235
	7.960329	3.305003	7.666752	2.882301
Average of avg and std	6.563097286	2.602200857	4.660397286	1.679247143
Std of avg and of std	2.153387609	0.625177479	1.988470385	1.130389678

We compute the standard deviation on the average execution time of the hoard memory allocator (0.625177479) we will see that it is smaller than the standard deviation of the average execution time of the standard memory allocator (1.130389678). The same applies to the standard deviation of the standard deviation computed by the *linux-scalability*.

Task 4 – Diving to the code – a big chunck memory allocator

We have also supplied a simple source code of the maman 12 solution with missing 350 lines (that were deliberately removed from the code). The code we have supplied enables allocating big chunks of memory and is located in the file big_chanks.c.

You can change the Makefile to compile the bct target with libmtmm.a library. The change is as following:

MYLIBS = libmtmm.a

#MYLIBS = libmtmmSSol.a

The bct (big chunk test) target will be compiled with libmtmm.a and you are encouraged to see in the code how big chuck allocation works. The big picturle is explained bellow:

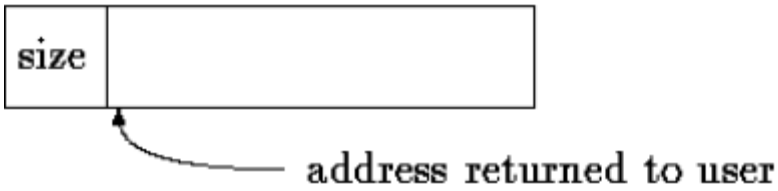
Consider using the mmap to allocate virtual memory for superblocks and large (more then S/2) memory chunks.

```
int fd = open("/dev/zero", O_RDWR);
p = mmap(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
close(fd);
```

```
or shortly
p = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1,
0);
```

Note: don't forget to check return values!

Since munmap requires size parameter, one has to save the size of the allocated memory. This size can be saved as depicted here:



Task 5 – Diving deeper in the code - Implement small chunk allocation

Following is the list of the files we have supplied:

- big-chanks.c – driver to test big chunk allocation
- linux-scalability.c – driver to test allocation of arbitrary size chunks from multiple threads
- core_memory_allocator.c – allocates and free memory from the system by mapping and unmapping anonymous files to virtual memory
- cpu_heap.c – implements functions that performs various operations at a single CPU heap level
- memory_allocator.c - the central module of memory allocator implementing malloc, free and a few helper functions
- size_class.c - implements functions to perform various operations at size class level such as insertions, removal and searches for superblocks from a given sizeclasss
- memory_allocator.h – helper functions for memory_allocator.c
- mtmm.h – malloc, free, realloc, calloc API and Hoar allocator constants
- ptbarrier.h – synchronization primitives
- Makefile – script to compile targets

You will find several functions left unimplemented and if you will try to link the libmtmm.a with linux_scalability driver segmentation fault will be the result (linux_scalability uses pthreads library that calls to the malloc and malloc for small chunks will return null).

So you need to implement the missing staff. Is about 350 lines of code (that we have removed).

References

[1]
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

[2]
"The C Programming Language" by Kernighan and Ritchie (attached to maman12.zip as a pfd)

[3]
http://en.wikibooks.org/wiki/C_Programming/C_Reference/stdlib.h/malloc#Implementations

[4]
http://en.wikipedia.org/wiki/Hoard_memory_allocator

מטלת מנחה (ממ"ן) 13

הקורס: "מערכות הפעלה"

חומר הלימוד למטלה: ראו פירוט בסעיף "רקע"

מספר השאלות: 5 **משקל המטלה: 12**

סמסטר: 2016א **מועד אחרון להגשה:** 21.01.2016

הגשת המטלה: שליחה באמצעות מערכת המטלות המקוונת באתר הבית של הקורס.
הסבר מפורט ב"נוהל הגשת מטלות המנחה".

החלק המעשי (80%)

כללי

בחלק המעשי נכתוב שתי תוכניות קטנות המממשות פונקציונאליות של מנהל מערכת קבצים ext2 על דיסקט 1.44 Mb.

מטרה

הכרת מערכת קבצים ext2

רקע

א) פרק 13.2 ב Glibc manual המתייחס לפונקציות lseek, open, close, read, write.

ב) קובץ ext2.pdf/rtf המתאר את ה layout של ext2.

ג) The second extended file system המכיל פירוט של כל הקבועים, המבנים והאלגוריתמים. חלק זה נועד לסימוכין ולא נדרש לקרוא אותו במלואו.

תיאור המשימה

עליכם לכתוב 2 תוכניות:

my_dir <absolute_path_to_directory_residing_on_the_floppy_disk>

ו

my_rm < absolute_path_to_directory_or_file_residing_on_the_floppy_disk >

אשר הראשונה מביניהן (my_dir) מדפיסה את תוכן הספרייה שנתיבה מצוין כפרמטר התוכנית. התוכנית השנייה מבצעת מחיקה של הספרייה או של הקובץ שנתיבם מצוין כפרמטר התוכנית.

תוכנית my_dir

1) בעקבות הרצתה של הפקודה my_dir < *abs_path_to_directory_residing_on_the_floppy_disk* >, התוכנית תבדוק אם הספרייה נמצאת במערכת הקבצים ext2 אשר על גבי הדיסקט. במילים אחרות my_dir בודקת האם נתיב המצוין כפרמטר של התוכנית my_dir נמצא על הדיסקט.

3) לשם פשטות נניח שהפרמטר dir_name מציין נתיב מלא המתחיל ב /. לדוגמא, אם בשורש של ext2 נמצאת ספרייה בשם a ובתוכה נמצאת ספריה b, אז הרצת הפקודה my_dir /a/b, תסתיים בהצלחה והתוכנית תדפיס את תוכן הספרייה ותחזיר סטאטוס 0.

4) הפורמט המודפס חייב להיות זהה לפורמט של פתרון ביה"ס.

5) במידה והנתיב *abs_path_to_directory_residing_on_the_floppy_disk* אינו נמצא על ה ext2, התוכנית my_dir תודיע הודעת שגיאה ותחזור עם סטאטוס 1.

5) במקרה של כישלון של קריאת מערכת כלשהי התוכנית תחזור עם סטאטוס 1.

תוכנית my_rm

1) בעקבות הרצה של my_rm < *abs_path_to_dir_or_file_residing_on_the_floppy_disk*> התוכנית תמחק את הקובץ (או את הספרייה עם כל תכולתה) ממערכת הקבצים ext2.

2) כמו במקרה של התוכנית my_dir, גם כאן נניח שהנתיב הוא נתיב יחסית לשורש של ext2 ושהנתיב מתחיל ב /.

3) אם המחיקה הצליחה, התוכנית my_rm תחזור עם סטאטוס 0. אחרת התכנית תחזיר סטאטוס 1.

4) במקרה של כישלון של קריאת מערכת כלשהי, התוכנית תחזור עם סטאטוס 1.

קראו בעיון את הקובץ ext2.pdf/rtf. מסביר את מהו layout של מערכת הקבצים ext2. במידת הצורך תוכלו לעיין ב The second extended file system המכיל פירוט של כל הקבועים, המבנים והאלגוריתמים של ext2.

הגשה

יש להגיש קבצי קוד וקובץ Makefile שמייצר קבצי הרצה בשם my_rm ו my_dir. אין להגיש קבצים מקומפלים. את הקבצים המוגשים יש לשים בקובץ ארכיון בשם exYZ.zip (כאשר YZ הנו מספר המטלה). הכנת קובץ ארכיון מתבצעת ע"י הרצת הפקודה הבאה משורת הפקודה של Linux:

zip exYZ.zip <ExYZ files>

<u>הערה חשובה: בכל קובץ קוד שאתם מגישים יש לכלול כותרת הכוללת תיאור הקובץ, שם הסטודנט ומספר ת.ז.</u>

<u>פתרון ביה"ס</u>

קיבלתם את התוכניות my_rm ו my_dir כפי שמומשו על ידינו. שימו לב שאתם צריכים כונן דיסקטים וירטואלי עם מערכת הקבצים ext2. להכנת הדיסקט בצעו את השלבים הבאים:

1) save floppy.iso (from maman13.zip) to some folder on Ubuntu. E.g. in /tmp/floppy.iso
2) sudo rm /dev/fd0
3) sudo ln -s /tmp/floppy.iso /dev/fd0
4) sudo losetup /dev/loop0 /tmp/floppy.iso
5) sudo mkdir /media/floppy
6) sudo mount /dev/loop0 /media/floppy

כעת /dev/fd0 יהיה מקושר לקובץ floppy.iso. ו /media/floppy יכיל את הקבצים והספריות הבאות:

a b foo1 foo2 foo3 lost+found

ואם תריצו את my_dir תקבלו:

```
27-Jun-2009 16:34 .
27-Jun-2009 16:34 ..
27-Jun-2009 16:00 lost+found
27-Jun-2009 16:33 a
27-Jun-2009 16:33 b
27-Jun-2009 16:33 foo1
27-Jun-2009 16:33 foo2
27-Jun-2009 16:34 foo3
```

אתם יכולים גם ליצור ספריות וקבצים חדשים בדיסקט ע"י יצירתם ב /media/floppy. אל תשכחו להריץ sync כדי לגרום למערכת הפעלה לכתוב את השינויים בפועל על גבי הדיסקט הווירטואלי. אחרת התוכנית my_dir "תראה" את השינוי כעבור דקה כאשר מערכת ההפעלה תחליט לשמור את ה buffer cache.

כדי שתוכלו להריץ את הפיתרון, יש לוודא שהרשאת x במחרוזת ההרשאות של הקבצים של פיתרון בה"ס נמצאות במצב "דלוק". כדי "להדליק" אותה במידה והיא "כבויה" יש להריץ משורת הפקודה של UNIX את הפקודה:

```
chmod +x my_rm my_dir
```

החלק העיוני (20%)

שאלה 1 (5%)

השוו בין pooling ו interrupt-driven I/O. תארו מצב אחד שבו עדיף להשתמש ב pooling על פני interrupt-driven I/O ומצב אחד שעדיף להתמש ב interrupt-driven I/O על פני pooling.

שאלה 2 (5%)

מערכי דיסקים RAID level 2 ו RAID level 3 מסוגלים להמשיך לעבוד כאשר אחד מהדיסקים במערך מתקלקל. יחד עם זאת, Level 2 דורש מספר רב יותר של דיסקים עודפים. אז מדוע יש בכלל עניין כלשהו בשיטה הזאת?

תזכורת - קוד המינג:

בהינתן מילה בת 4 סיביות:

סיבית b1	סיבית b2	סיבית b3	סיבית b4

קוד המינג שלה הוא:

P1	P2	B1	P3	B2	B3	B4

כאשר

P1 = Even Parity of b1, b2, b4

P2 = Even Parity of b1, b3, b4

P3 = Even Parity of b2, b3, b4

לדוגמא: המינג קוד של מילה בת 4 סיביות (משמאל לימין – מ least significant bit ל most significant bit) 1101 יהיה 1100110.

שאלה 3 (5%)

בשיטת ה i-nodes, אם תסתכלו על כמות הבלוקים שניתן להצביע עליהם באמצעון מיעון עקיף מרמה שלוש (triple indirection), אפשר להבחין שכמות הבלוקים תהיה כמעט זהה לכמות הבלוקים עליהם ניתן להצביע באמצעות מיעון ישיר + כמות הבלוקים עליהם ניתן להצביע באמצעות מיעון עקיף מרמה אחת + כמות הבלוקים עליהם ניתן להצביע באמצעות מיעון עקיף מרמה שניה + כמות הבלוקים עליהם ניתן להצביע באמצעות מיעון עקיף מרמה שלישית. אז מדוע אין זה רעיון טוב להשתמש רק במיעון עקיף מרמה שלשוש?

שאלה 4 (5%)

תארו את שיטת ה capabilites ל domain control.

הגשת החלק העיוני

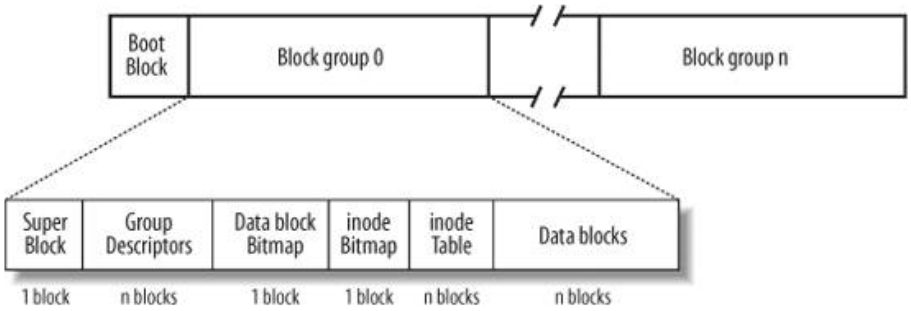
החלק העיוני יוגש כקובץ Word במערכת הפעלה Windows. שם הקובץ צריך להיות exYZ.doc (כאשר YZ הנו מספר המטלה).

The EXT2 file system

Partialy based on http://www.tldp.org/LDP/tlk/fs/filesystem.html

1. Physical layout of the ext2 file system

The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created. Every file's size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.



The figure above shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks which can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver and/or firmware code run by the device controller. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks[1]. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is necessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

[1] In our exercise we just "open" the /dev/fd0 and perform series of size of block byte "read"s.

2. The superblock

The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

Magic Number
: This allows the mounting software to check that this is indeed the Superblock for an EXT2 file system.

Revision Level
: The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system,

Mount Count and Maximum Mount Count
: Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message ``maximal mount count reached, running e2fsck is recommended'' is displayed,

Block Group Number
: The Block Group number that holds this copy of the Superblock,

Block Size
: The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group
: The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks
: The number of free blocks in the file system,

Free Inodes
: The number of free Inodes in the file system,

First Inode
: This is the inode number of the first inode in the file system. The first inode in an EXT2 root file system would be the directory entry for the '/' directory.

Above information is contained in the struct ext2_super_block. Partial list of the ext2_super_block fields are depicted bellow:

```
struct ext2_super_block {
        __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
        __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_frag_size;        /* Fragment size */
        __le32  s_blocks_per_group;     /* # Blocks per group */
        __le32  s_frags_per_group;      /* # Fragments per group */
        __le32  s_inodes_per_group;     /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
        __le32  s_wtime;                /* Write time */
        __le16  s_mnt_count;            /* Mount count */
        __le16  s_max_mnt_count;        /* Maximal mount count */
        __le16  s_magic;                /* Magic signature */
        __le16  s_state;                /* File system state */
        __le16  s_errors;               /* Behaviour when detecting errors */
        __le16  s_minor_rev_level;      /* minor revision level */
        __le32  s_lastcheck;            /* time of last check */
        __le32  s_checkinterval;        /* max. time between checks */
        __le32  s_creator_os;           /* OS */
        __le32  s_rev_level;            /* Revision level */
        __le16  s_def_resuid;           /* Default uid for reserved blocks */
        __le16  s_def_resgid;           /* Default gid for reserved blocks */
```

Other fields are irrelevant to us right now. Use the #include <linux/ext2_fs.h> preprocessor directive in order to obtain the struct ext2_super_block definition. __le16, __le32 are defined in <linux/types.h> (includes path on our Knoppix machines is located in /usr/include) and denote the little-endian ordering for words and double-words (the least significant byte is stored at the highest address).

The `s_inodes_count` field stores the number of inodes, while the `s_blocks_count` field stores the number of blocks in the Ext2 filesystem.

The `s_log_block_size` field expresses the block size as a power of 2, using 1,024 bytes as the unit. Thus, 0 denotes 1,024-byte blocks, 1 denotes 2,048-byte blocks, and so on.

The `s_blocks_per_group` and `s_inodes_per_group` fields store the number of blocks and inodes in each block group, respectively.

3. Mapping disk data directly to structures

For this exercise, you are to use devise interface as thought it is a simplified block device interface. That is, you are to open the file ("/dev/fd0") and then read or write its contents as full ext2 blocks. Use the following functions to read the disk.

```
int fid; /* global variable set by the open() function */
int block_size;     /* bytes per sector from disk geometry */
…
fid = open ("/dev/fd0", O_RDWR);
block_size = /* read from the superblock */

int fd_read(int block_number, char *buffer){
        int dest, len;
        dest = lseek(fid, block_number * block_size, SEEK_SET);
        if (dest != block_number * block_size){
              /* Error handling */
        }
        len = read(fid, buffer, block_size);
        if (len != block_size){
                /* error handling here */
        }
        return len;
}
```

Then copy from buffer to desired structure. See, for example, how to read in to the super block structure:

```
struct ext2_super_block sb;

memcpy(sb, buffer, sizeof(struct ext2_super_block);
```

4. The EXT2 Group Descriptor

Each Block Group has a data structure describing it. Like the Superblock, all the Group Descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption.

Each Group Descriptor contains the following information:

Blocks Bitmap
The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation,

Inode Bitmap
The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,

Inode Table
The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.

Free blocks count, Free Inodes count, Used directory count

The group descriptors are placed on after another and together they make the Group Descriptors Table. Each Blocks Group contains its copy of the entire Table of Group Descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

An `ext2_group_desc` (from /usr/include/linux/ext2_fs.h) corresponds for Group Descriptor structure.

```
struct ext2_group_desc
{
        __le32  bg_block_bitmap;                /* Blocks bitmap block */
        __le32  bg_inode_bitmap;                /* Inodes bitmap block */
        __le32  bg_inode_table;         /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count;     /* Directories count */
```

The `bg_free_blocks_count`, `bg_free_inodes_count`, and `bg_used_dirs_count` fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure.

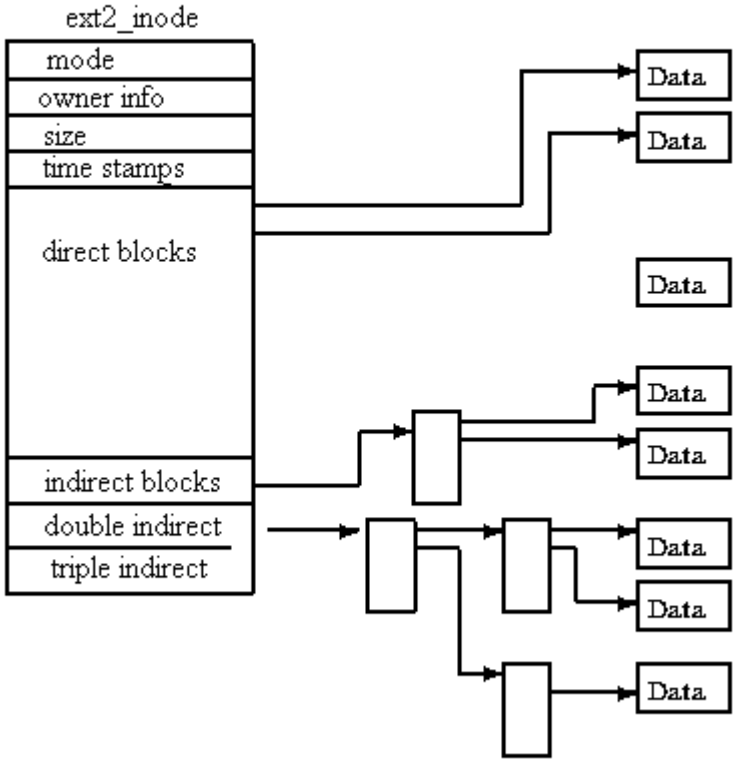
5. Data block bitmap and inode bitmap

The bitmaps are sequences of bits, where the value 0 specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Because each bitmap must be stored inside a single block and because the block size can be 1,024, 2,048, or 4,096 bytes, a single bitmap describes the state of 8,192, 16,384, or 32,768 blocks.

6. Inode table

In the `EXT2` file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode

table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure bellow shows the EXT2 inode:



Amongst other information, it contains the following fields:

mode
: This holds two pieces of information; what does this inode describe and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information
: The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size
: The size of the file in bytes,

Timestamps
: The time that the inode was created and the last time that it was modified,

Datablocks
: Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

An `ext2_inode` (from linux/ext2_fs.h) corresponds for Inode structure:

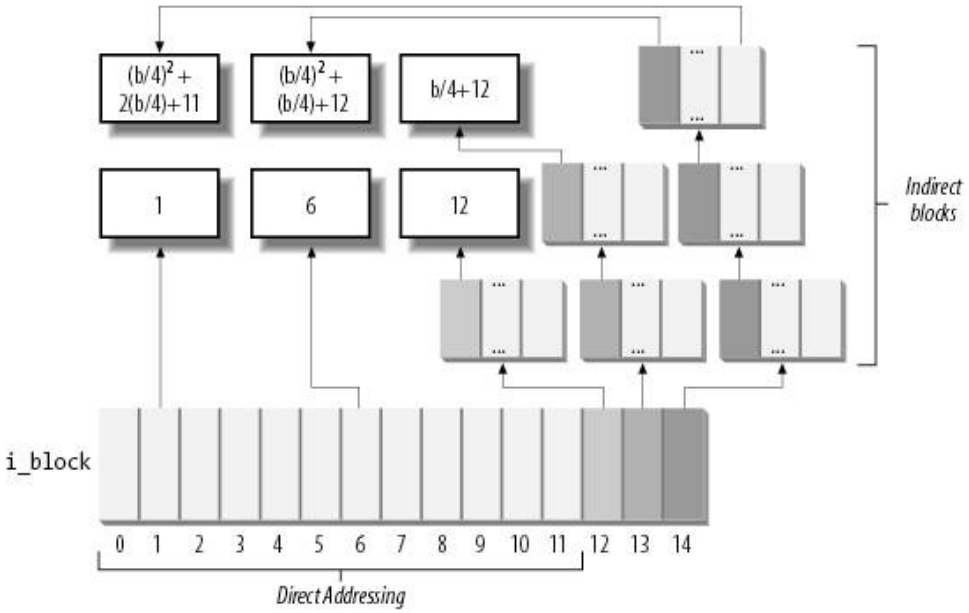
```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_reserved1;
                } linux1;
                struct {
                        __le32  h_i_translator;
                } hurd1;
                struct {
                        __le32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
```

`osd1` union field that comes after the `i_flags` is irrelevant to us right now (it contains specific operating system information).

The `i_size` field stores the effective length of the file in bytes, while the `i_blocks` field stores the number of data blocks that have been allocated to the file.

The values of `i_size` and `i_blocks` are not necessarily related. Because a file is always stored in an integer number of blocks, a nonempty file receives at least one data block and `i_size` may be smaller than `(size of block)` * `i_blocks`. On the other hand, by applying `lseek` a file may contain holes. In that case, `i_size` may be greater than `(size of block)` * `i_blocks`.

The `i_block` field in the disk inode is an array of `EXT2_N_BLOCKS` components that contain logical block numbers. In the following discussion, we assume that `EXT2_N_BLOCKS` has the default value, namely 15. The array is illustrated in:

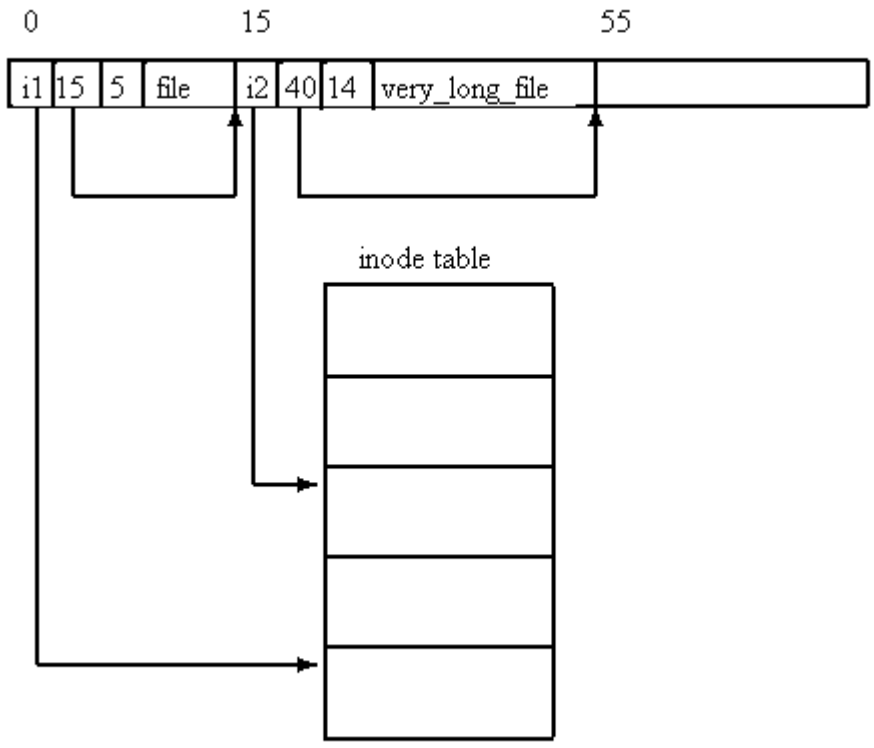


As can be seen in the figure, the 15 components of the array are of 4 different types:

- The first 12 components yield the logical block numbers corresponding to the first 12 blocks of the file to the blocks that have file block numbers from 0 to 11.
- The component at index 12 contains the logical block number of a block, called indirect block, that represents a second-order array of logical block numbers. They correspond to the file block numbers ranging from 12 to b/4+11, where b is the filesystem's block size (each logical block number is stored in 4 bytes, so we divide by 4 in the formula). Therefore, the one must look in this component for a pointer to a block, and then look in that block for another pointer to the ultimate block that contains the file contents.
- The component at index 13 contains the logical block number of an indirect block containing a second-order array of logical block numbers; in turn, the entries of this second-order array point to third-order arrays, which store the logical block numbers that correspond to the file block numbers ranging from b/4+12 to $(b/4)^2+(b/4)+11$.
- Finally, the component at index 14 uses triple indirection: the fourth-order arrays store the logical block numbers corresponding to the file block numbers ranging from $(b/4)^2+(b/4)+12$ to $(b/4)^3+(b/4)^2+(b/4)+11$.

7. EXT2 Directories

In the EXT2 file system Directories are special files that are used to create and hold access paths to the files in the file system. This figure shows the layout of a directory entry in memory:



A directory file is a list of directory entries, each one containing the following information:

inode
: The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group. In figure, the directory entry for the file called `file` has a reference to inode number `i1`,

name length
: The length of this directory entry in bytes,

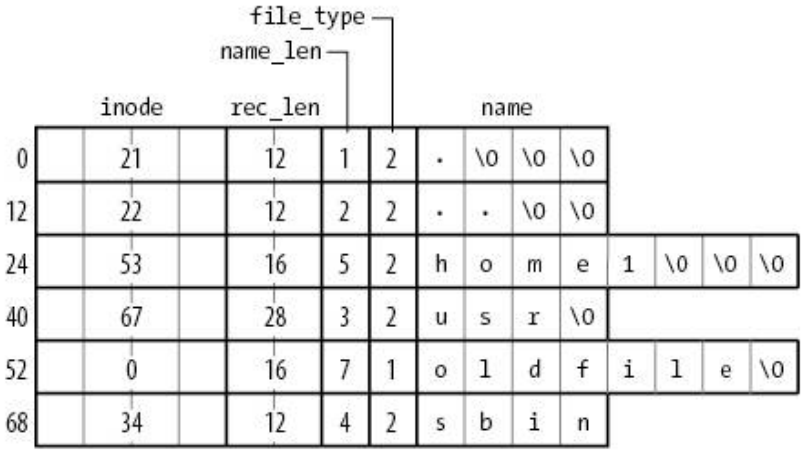
name
: The name of this directory entry.

The first two entries for every directory are always the standard ``." and ``.." entries meaning ``this directory" and ``the parent directory" respectively.

An `ext2_dir_entry_2` (from linux/ext2_fs.h) corresponds for directory structure:

```
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

The structure has a variable length, because the last `name` field is a variable length array of up to `EXT2_NAME_LEN` characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters (`\0`) are added for padding at the end of the filename, if necessary. The `name_len` field stores the actual filename length.



The `file_type` field stores a value that specifies the file type. The `rec_len` field may be interpreted as a pointer to the next valid directory entry: it is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its `inode` field to 0 and suitably increment the value of the `rec_len` field of the previous valid entry. **Read the `rec_len` field of carefully; you'll see that the *oldfile* entry was deleted because the `rec_len` field of *usr* is set to 12+16 (the lengths of the *usr* and *oldfile* entries).**

8. Finding a Directory/File in an EXT2 File System

A Linux filename has the same format as all Unix filenames have. It is a series of directory names seperated by forward slashes (``/'') and ending in the file's name. Let's call them **path components** or **direntries**. One example filename would be `/home/rusling/.cshrc` where `/home` and `/rusling` are directory names and the file's name is `.cshrc`. Like all other Unix systems, Linux does not care about the format of the filename itself; it can be any length and consist of any of the printable characters. To find the inode representing this file within an `EXT2` file system the system must parse the filename a directory at a time until we get to the file itself.

The first inode that we need is the inode for the root of the file system and we find its number in the file system's superblock. To read an EXT2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 2 then we need the 2nd inode from the inode table of Block Group 0. The root inode is for an EXT2 directory, in other words the mode of the root inode describes it as a directory and it's data blocks contain EXT2 directory entries.

`home` is just one of the many directory entries and this directory entry gives us the number of the inode describing the `/home` directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the `rusling` entry which gives us the number of the inode describing the `/home/rusling` directory. Finally we read the directory entries pointed at by the inode describing the `/home/rusling` directory to find the inode number of the `.cshrc` file and from this we get the data blocks containing the information in the file.

9. Approaching the maman13

1) You have to submit my_dir and my_rm executables. We have supplied a code of my_dir (see the my_dir.c file). So first of all take a look at the code and make sure you understand whats going on. You can also run my_dir step by step in debugger (this code runs in user space).

2) In order to run my_dir, fisrt setup the virtual floppy as described by "Pitaron Betsefer" paragraph in maman13.doc. Following the description will "connect" the image of the 1.44 MB ext2 file system to the /dev/fd0. /dev/fd0 is a name of the floppy drive on the Linux operating system and my_dir program expects the existence of the floppy drive with ext2 file system on it at /dev/fd0.

3) Once you understand the my_dir.c code and have run it on the virtual floppy drive, you are ready to implement the code of the my_rm utility. The implementation takes 250 lines of the code. The only difference now, is that you need additional functions to write an updated i-node, free blocks bitmap etc back to the ext2 layout. Take a look at the code of the my_rm.c and fill the missing.

4) The big picture as as following: my_rm gets a path to the file name or to the directory name. Then the code starts to traverse the extfs file system. During the traversal (invoked from the ***rdump_inode*** function), 2 stacks are filled as described bellow:

```
// call recursive fuction that makes the traversal and poplates the stacks
// if file name is specifies to the my_rm, them only fileNamesStack is populated
// if dir name is specified to the my_dir, then dirNamesStack is populated by at leat one entry
// and if the directory to be removed is not empty, additional entries are inserted to file and/or directory stacks
rdump_inode(ino, &inode, p, "");
```

When the stack (or stacks) are populated by the path names, the files (if any file exists) are removed one by one and then directories are also removed one by one.

```
while (!isEmpty(fileNamesStack))
 {
      printf("We popped from fileNamesStack: %s\n", (tmp = (char*)pop(fileNamesStack)));
      remove_file_by_name(tmp);
      free (tmp);
 }
 while (!isEmpty(dirNamesStack))
 {
      printf("We popped from dirNamesStack: %s\n", (tmp = (char*)pop(dirNamesStack)));
      remove_directory_by_name(tmp);
      free(tmp);
 }
```

The implementation is the easiest one to debug since:

1) the directories are already empty at the time of removal
2) removing names from the stack facilitates that first the innermost directories are removed.

So the general structure is supplied in the my_rm.c. All you have to do is to implement several low level functions that updates i-node, free blocks bitmap (that are missing in my_rm.c).

Enjoy!

References

1)
http://www.tldp.org/LDP/tlk/fs/filesystem.html

2)
http://www.nongnu.org/ext2-doc/ext2.html