בס"ד יא' חשוון תשע"ד

**תכנות מתקדם- סיכומי הרצאות**

**שיעור 1:**

**שקופית 7:** לשנן כמה תופס כל משתנה. ההבדל בין sign לunsign הוא הביט הראשון האם הוא מושקע בסימן או לא (ואם לא נוכל להגיע ל2^7).

מה ההבדל בין מערכת 32 ו64 ביט? מערכת ההפעלה צריכה שמור כתובות של משתנים כדי לדעת לגשת אליהם. מערכת 32 ביט היא מערכת בה צריך 32 ביטים כדי לייצג כתובת כלומר מספיק 4 בתים כדי לגשת לכתובת (ולכן אם יש יותר מ4 ג'יגה בזיכרון- לא נוכל לגשת אל הזיכרון הזה כי אין לנו דרך לפנות לכתובות.

במערכת 64 ביט, שומרים 8 בתים לכתובת. כלומר צריך שילוב בין החומרה שיהיה בה מספיק זיכרון ובין מערכת ההפעלה שיכולה לגשת למספר כתובות מסוים.

היסטוריה: המחשב הראשון נבנה במלחמת העולם השניה. לקחו 10 שנים עד שהומצא הקומפיילר ועד אז דיברו בשפה בינארית.

תהליך הקומפלציה מורכב בהתחלה משלב קדם מעבד (pre-processor) בו מתבצעות פעולות המתחילות ב#. מושתל קוד איפה שיש include וכדו'. בשלב זה יכולים להיות באגים שלא נמצא הקובץ שצריך לשתול וכו'.

**שקופית 14-** שלבים בקומפלציה:

1. הלקסר מקבל את הקובץ ומתחיל לקרוא את התוים שבו. הוא מעביר את התוים לtokenizer שבודק האם הגענו לסוף מילה ואם כן, הוא סוגר את המילה ביחידת token. אופרטורים נסגרים גם הם כיחידת token. הtoken מעיף את הרווחים, שורות חדשות וכו', ובונה למעשה מערך של tokens. באג שיכול להיווצר בשלב זה הוא בהופעת תוים לא מוכרים. הtokenizer גם מאפיין את הtokens מאיזה סוג תחבירי הם (int, מילים וכו').
2. הסיטקס בודק שהתחביר של הtokens תקין. בשלב זה יכולים להיות באגים כשאין ; וכדו'. תוך כדי קריאה, כל שם משתנה או פונקציה נכנס לטבלת symbol table בה נשמר איזה סוג משתנה הוא (גלובלי, לוקאלי וכדו') ומה סוגו (שם משתנה או פונקציה) וכתובתו. לכן אם יש שימוש במשתנה שלא הוצהר, הקומפיילר יתעצבן, ואם זה משתנה שיש בעיה בגישה אליו הלינקר יכעס.
3. עד עכשיו בנינו משפטים ועכשיו צריך לבדוק את הסמנטיקה בה נעלה על error כמו פונקציה שצריכה להחזיר ערך ולא עושה זאת/ השמה בין פוינטרים מסוג שונה. התחביר נכון ולכן עד עכשיו לא עלינו על השגיאה אך הסמנטיקה שגויה ולכן נעלה עליה בשלב זה.
4. רוב הקומפיילרים עושים בשלב זה אופטימיזציה- בודקים אם יש משתנים מיותרים שאפשר להעיף ולשנות מעט את הקוד. דוגמאות: חישובים שכבר בוצעו ישתלו במקום להיות מחושבים פעמים (שקופית 24), הקומפיילר מבין על איזה מעבד הוא רץ ולכן אם הוא יכול לבצע 4 פעולות במקביל, הוא יוכל לפתוח לולאה שרצה 4 פעמים ל4 פעולות במקביל (שקופית 27).
5. קיבלנו סט פקודות. עד עכשיו הקומפיילר קימפל רק קבצי cpp אך לא קבצי h שיקראו רק עם נעשה להם include מcpp ואז בשלב קדם מעבד הם יועתקו לcpp. החלוקה לקבצים חוסכת בזמן. אחרי שהקומפיילר קורא קובץ, הוא יוצר ממנו קובץ .obj. הלינקר יגש לקבצי .obj ויעשה להם השלמות לאיזה כתובת לגשת לפי טבלת הסימבל. בשלב זה נוצר קובץ ריצה שהוא קובץ exe. באג לינקר קורא כשהכתובת לא נמצאה- יש הצהרה בלי מימוש.

Lib dll- עוד סוגי תוצרים של הלינקר. אפשר כך לעבוד בשילוב בין חברות בשלב הלינקין בלי שילוב בשלבים שלפני כן בקומפלציה.

לקובץ שהורץ נוסף בהתחלה מידע איך הורץ, על איזה מעבד וכו'.

Interperter- בניגוד לקומפיילר, הוא לא מייצר קובץ ואז מריץ אותו, אלא מתרגם ומריץ שורה שורה. החיסרון בכך הוא האיטיות, התלות בקוד כדי להריץ את התוכנה כי אין קובץ ריצה, אין אפשרות לאופטימיזציה כי לא עוברים על הקוד בהקשר. היתרון הוא שבניגוד לקובץ ריצה המתאים למכונה ספציפית ולכן לא יכול לעבור בין מחשבים- interperter מאפשר עבודה בכל סביבה עם interperter המתאים לה.

ב java החליטו להריץ בהתחלה את כל מה שאפשר להריץ בלי תלות במכונה, ונוצר קבצי class חצי מקומפלים. אח"כ כשנריץ במחשב את הקבצים הללו, בעזרת JVM תוך כדי ריצה עושים לינקינג ספציפי למערכת ההפעלה שלנו. הוסיפו את הJIT שעובד יחד עם הinterperter וקורא קדימה כדי לשפר את הריצה ולעשות אופטימיזציה כדי לחסוך בזמן ריצה תוך כדי ריצה. כך מקבלים קוד שאפשר להריץ בכל מחשב אם יש לו virtual machine.

**שיעור 2:**

בזיכרון יש לנו את הstack והheap. איך זה עובד?

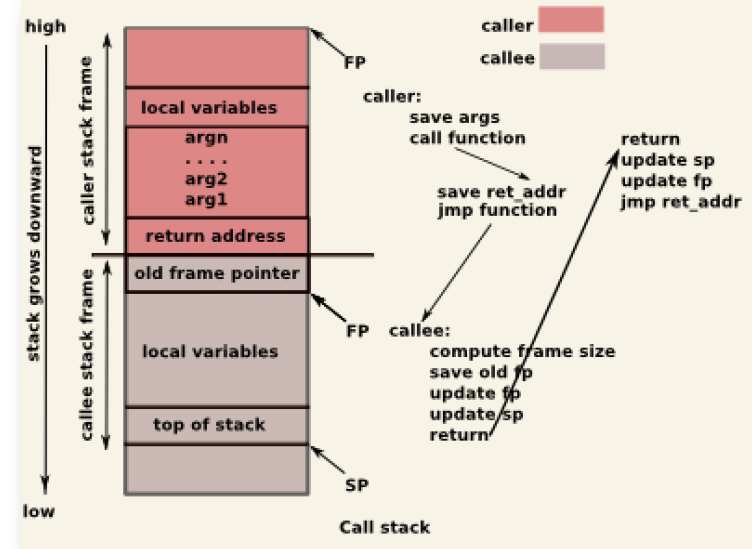
תוכנית שרצה תופסת מקום בזיכרון. יש את הDATA שהוא read only , הסטטים והגלובלים נמצאים בDATA בנפרד משאר המשתנים כי הם צריכים להיות נגישים כל הזמן ולכן לא יכולים להיות במחסנית עם שאר המשתנים.

בהמשך יש heap וstack. על הstack הקומפיילר יודע כבר מה יהיה ולעיתים ממלא את הזיכרון שבו לפני שהתוכנית בכלל רצה. הheap לרוב ניתן בגודל קבוע, והstack ניתן לפי הערכה של הקומפיילר כמה גודל נצטרך. היחסי גדלים בין הheap והstack לא קבועים ובחלקים שונים של התוכנית יש דינאמיות בגבול ביניהם.

***הstack*** שטח זיכרון רציף שהוא LIFO. מחסנית. יש רכיב במעבד בו נשמר ראש המחסנית וכך ידוע ע"י בקשה של הרגיסטר להיכן להכניס משתנים.

רגיסטר- זיכרון הכי מהיר שיושב על המעבד.

בהתחלה מוכנס למחסנית כתובת למקום בקוד שלפני הקריאה לפונקציה, שומרים מקום לערך מוחזר לפי סוג הפונקציה, jump- פקודת קפוץ לפונקציה, מיקום ארגומנטים, שמירת ערכי הפונקציה בFRAME שלה- במצגת לפי סדר מה מוכנס.



מתחילים מחסנית מלמעלה למטה (הheap מלמטה למעלה וכך הם מתקדמים זה לקראת זה).

Frame pointer- תחילת השטח על הstack של הפונקציה הנוכחי.

בקריאה לפונקציה, נשלחים ארגומנטים, כותבים return address שנדע אח"כ לאן לחזור ואז מבצעים jump לפונקציה חדשה. לפני הjump צריך לשמור מי היה הFP כדי שכשנחזור נדע היכן הFP של הפונקציה הקודמת שקראה לנו.

כשSP, הפוינטר איתו מטיילים בSTACK מגיע לFP יודע שהסתיימה הפונקציה ומעדכן לFP הקודם שנשמר.

הreturn address מדבר על המקום בקוד אליו חוזרים והוא יהיה בסוף FRAME בניגוד לFP שמכוונן למיקומנו על המחסנית.

* יש אפשרות להריץ תוכניות בדיבג או בrelease. בהרצה בדיבג יש שמירה של משתנים, טבלאות וכתובות המשולבות לחילופין עם הקוד כדי לאפשר ביצוע פקודה פקודה יחד עם אופטימיזציה ושאר הדברים הדורשים הקשר כשהקומפיילר מקמפל. לכן הקובץ גדול בהרבה מrelease בו אי אפשר לעצור באמצע. ע"י קריאה באסמבלר אפשר לראות מה המחשב מבצע בהרצת תוכנית.

כשפונקציה היא inline, נחסך כל הסיפור והפרוצדורה עם הstack לעיל.

Stack overflow- הקומפיילר הרבה פעמים לא יעלה על זה בקומפלציה אלא רק כשיתחיל למשל קריאות רקורסיביות בזמן ריצה. זה יקרה כשהמחסנית מלאה. בד"כ הבעיה היא שהיינו צריכים להעביר דברים לheap שיש בו הרבה יותר מקום.

יתרונות מחסנית- מהירה כי הכל יושב רציף, מנוהלת ע"י הקומפיילר ביצירת ומחיקת אובייקטים.

חסרון מחסנית - קטנה.

***Heap***:

יש שימוש בnew להקצאה והגודל לא ידוע מראש. בד"כ מצביע נמצא על המחסנית ומנוהל על הקומפיילר, והוא מצביע לזיכרון שבheap.

**ניהול זיכרון:**

מדבר על ניהול זיכרון בheap כי את הstack הקומפיילר מנהל. יש שפות שדורשות יותר ניהול ממנו. רוצים לעשות שימוש חוזר בזיכרון שכבר הוקצה כדי שהזכרון לא יגמר. לכן מחפשים מקום פנוי.

ניהול זיכרון בד"כ מדבר על 3 רמות:

1.חומרה. RAM ולא על דיסק קשיח

2. מערכת ההפעלה

3. מה אנחנו יכולים לעשות ליעל תוכנית.

בחומרה, יש דיסק קשיח (נראה כמו פטיפון) ויש RAM שנראה כמו דיסק און קי. ככל שזיכרון קרוב יותר למעבד, הוא קטן, מהיר ויקר יותר. זיכרון רחוק איטי יותר כי הוא לא על לוח האם אלא המידע אליו זורם דרך כבל. בד"כ הוא גדול יותר כי הוא לא על לוח האם והוא זול יותר.

מהקרוב למעבד לרחוק: אוגרים=רגיסטרים (יושבים על המעבד), זיכרון מטמון=cache, ram=main memory, ודיסק קשיח. המעבד פונה למידע לידו ואם הדבר שמחפשים לא נמצא, לאט לאט נמשיך לרמות רחוקות (רואים את זה בוורד שהמחשב מצפה שקודם נקרא עמודים סמוכים ולכן אם נבקש עמוד רחוק- ייקח לו זמן להטען).

לנו כמתכנתים אין גישה לחומרה אלא אנו משתמשים במערכת ההפעלה שתשתמש בdriver כדי לגשת לחומרה.

מערכת ההפעלה דואגת שלכל תוכנית יהיה את הזיכרון שלה ולא תהיה גלישה של תוכנית על תוכנית אחרת בזיכרון. דבר נוסף לו דואגת מערכת ההפעלה הוא זיכרון וירטואלי. יש כתובות פיזיות (איפה מידע באמת יושב) ולוגיות (איפה הוא חושב שהוא יושב). בRAM יש זיכרון מוגבל ומערכת ההפעלה נותנת לנו הרגשה שיש לנו יותר זיכרון, זהו זיכרון וירטואלי. היא מנצלת מקומות שכרגע פנויים עד שיבקשו אותם ב"שיטת הכיסא החם". זה נקרא זיכרון וירטואלי. זיכרון וירטואלי הוא למעשה טבלה ולא באמת זיכרון. בטבלה זו ממופה לאיזה כתובת בזיכרון ממופה משתנה. ייתכן ש2 משתנים ממופים לאותו זיכרון כי הם לא יהיו עליו במקביל. אם מערכת ההפעלה נתקעת פתאום כי המקום באמת נגמר- היא יכולה לבקש שנסגור תוכניות. פתרון מגדיל ראש יותר הוא ע"י פתיחת קובץ swap file בדיסק הקשיח אליו מתייחסים כאל זיכרון בRAM והזיכרון וירטואלי ימפה גם אליו. התוצאה למשתמש היא שתוכניות יהפכו ליותר ויותר כבדות וההארד דיסק יעבוד ביתר אינטנסיביות.

לפעמים גם הswap file ייגמר ואז מערכת ההפעלה תתחנן שנסגור תוכניות.

איך המחשב מחלק איזה מידע לשים באיזה זיכרון?

משתנה שחיפשנו ומצאנו בתהליך חיפוש מתרחק מהמעבד בעבר, יישמר בcache.

ברגיסטרים מושמים משתנים שעכשיו עושים עליהם פעולות אריתמטיות וכדו.

כשבcache ייגמר המקום, נבדוק מי המשתנה הכי ישן בו- הוא יועף ובמקום שיתפנה נכניס משתנה חדש. (יש אלגוריתמים חכמים יותר לניהול הcache).

את כל הניהול הנ"ל עושה מערכת ההפעלה.

ברמת העבודה שלנו כמתכנתים, אנחנו רוצים לא לבזבז זיכרון. בעיות אפשריות:

1. פרגמנטציה- זו בעית חורים מסידור לא נכון של משתנים בזיכרון (בעיית חניה). נרצה להקצות דברים ברצף.
2. dangling pointer- פוניטר שמצביע לזבל.
3. Mempry leak- בעיה של הקצאת עוד ועוד זיכרון. נרצה שבתחילת תוכנית תהיה עליה בצריכת זיכרון ולאחר מכן כמות זיכרון קבועה בשימוש.
4. עקרון המקומיות- ההיגיון אומר שאם השתמשנו במשתנה נצטרך גם את אלה שבסביבתו. לכן מערכת ההפעלה אומרת שאם נטעין לcache משתנה- נטעין כבר בלוק שלם בסביבתו. הליכה על מטריצה למשל לפי עמודה, לא תנצל את מערך השורה שהועבר לcache ותכריח את מערכת ההפעלה להזיז שורות מהcache ודיסק הקשיח שוב ושוב.

בשבוע הבא נלמד כיצד לכתוב ניהול זיכרון.

**שיעור 3:**

רמות שונות של שליטה בזיכרון: הזזת זיכרונות לפינוי מקום-כנגד פרגמנטציה בה "החניה לא פנויה", נרצה לבצע דה-פרגמנטציה.שימוש בזיכרון ששוחרר לאחרונה (rcycling) ועוד.

בעבודה עם פוינטר היו לנו מספר בעיות:

פוינטר אחר ששחרר את הזיכרון שהפוינטר שלי מצביע עליו (dangling pointer) כך שנוצר מצב של פוינטר שמצביע לזבל. התוכנית תעוף רק ברגע שנרצה לפנות אל מה שיש בכתובת.

Inflexibale- ככל שאנחנו יותר ספציפיים למה שאנחנו מחפשים- נוכל לפעול בצורה מהירה יותר אך פחות כללית (למשל חיפוש המילה "לכבוד" במכתב שנחפש בהתחלה).

בגדול, יש לגשת בענווה למערכת ניהול הזיכרון של ג'אווה שכנראה יותר מוצלחת ממה שאנחנו נעשה (), אך בניהול מעט זיכרון לתוכנית ספציפית, יש מקום לניהול הזיכרון שלנו.

כדי שננהל כמו שצריך זיכרון, צריך לבצוע רישום מה הוקצה ומה שוחרר ופנוי (bookkeeping).

מערכת ניהול זיכרון אוטומטית, עובדת עם garbage collection שעובד כל הזמן ומנהל רישום כל הזמן כדי לבדוק את מי אפשר לשחרר בכל רגע נתון.

כשאנחנו מקצים זיכרון אנחנו רואים את הזיכרון כאוסף bytes אולם מערכת ההפעלה רואה את זbytes כמילים. המעבד גם הוא יודע לעבוד עם מילים- למשל עם 4 bytes שהם מילה. לכן לא יהיה יעיל לכתוב char שהוא byte אחד ואז int שהוא 4 bytes כך שיש גלישה למילה נוספת, והמעבד צריך להרכיב מחדש את הזיכרון כך שint יהיה בword אחד. הוא יעשה הזחה ל2 הwords ויחפוף אותם כמו בתמונה לעיל כדי שיוכל לקרוא את המידע:



לכן, יש חשיבות לסדר בו אנחנו כותבים משתנים.

יש מצב שהמערכת מבצעת alignment שזו בדיקה שמשתנים יתחילו במקום בו למעבד יהיה הכי קל לקרוא אותם. הdefault הוא aligment ל4 bytes.

כלל נוסף הוא שלמעבד קל לעבוד בכפולות של 2 (כי הוא מגיע אליהן ע"י ביצוע shift למספר בינארי שמשמעותה חלוקה/ הכפלה ב2) ולכן נשים משתנים בכתובות המתחלקות ב2/ תחילת word.

יוצא שמילוי זיכרון בצורה אופטימלית הוא ע"י הכנסה קודם של משתנים גדולים (הממלאים words שלמים), ואז בהדרגה משתנים קטנים יותר.

במילוי, תמיד יש מילוי לבלוקים שלמים, ולכן יושם ריפוד לחורים ולמה שנשאר עד word שלם, וכך אם ניצור מערך של stracts זה יהיה מסודר והדרך למיקום הזיכרון תשמר במקום לתכנן כל פעם מחדש איך לסדר אותו.

שקופית 59- דוגמא בה מתבזבזים 7 בתים בגלל הריפוד כדי שהמשתנים יהיו בכתובות זוגיות/ תחילת בית.

Maclloc- מקבל כפרמטר כמה בתים אנחנו רוצים להקצות.

New- מקבל כמה איברים אנחנו רוצים להקצות. (למעשה מאחורי הקלעים הוא מקבל כמה איברים ומאיזה סוג הם). נראה מדוע יש חשיבות להתאמת new ו delete תואם.

Delete new free malloc הן פונקציות מינימליות שלא בודקות בשבילנו אם דברים כבר שוחררו וכדו' ומנגנון מה יקרה בפונקציה תלוי בקומפיילר, והאחריות על המתכנת. בjava לקחו מהמתכנת את האחריות. זו בעיה שהיתה לנו כשהפונקציות לא שמרו עלינו.

בעיה נוספת היא שבמערכת real time צריך לנהל מתי דברים יקרו- מתי להתחיל לשחרר זיכרון ומתי להקצות בפינצטה ומתי לא. כשטיל עף- צריך שדברים יקרו מהר ולא נתחיל להתלבט איך הכי יעיל לנהל זיכרון. לכן new malloc ושאר החבר'ה לא ישמשו בreal time.

Malloc- דואג לaligment, מחזיר פוינטר לbyte הראשון שהוקצה.

כדי לממש malloc צריך לדעת מתי מתחיל ומסתיים הheap שבתוכו אנחנו מקצים את הזיכרון (manage\_memory\_start ). Last\_allocated, ישמור לנו מאיפה על הheap עוד לא הוקצה זיכרון.

בפונקציה malloc, הlast\_allocated שיוחזר יהיה size בתים הלאה בheap כאשר size גודל מה שהוקצה.

Free מורכב יותר. אנחנו נעשה פעולה סימטרית אם צריך לשחרר את הזיכרון האחרון שהוקצה אך מה עם שחרור משהו באמצע הזיכרון? גם malloc באופן שתואר לעיל הוא בעייתי כי הוא לא נעזר בחורים שעשויים להיווצר בזיכרון. נרצה להבין את free ודרכו נבין את malloc.

כדי לדעת כמה לשחרר כשאנחנו מגיעים לתחילת הקצאה של משתנה שרוצים לשחרר, נשמור בתחילת כל הקצאה header ובו הגודל שהקצאנו והאם הבלוק פנוי/ תפוס. Free, משמעותו תהיה להפוך את הheader המתאים מתפוס לפנוי. בmalloc נטייל על header בחיפוש מקום פנוי בקפיצות header לפי הsize הנמצא בheaders.

זה עדיין לא המימוש הכי יעיל כי אין פה ניצול של בלוקים סמוכים כזיכרון רציף, משתנה קטן יכול לתפוס בלוק גדול סתם ועוד.

השיטה שאנו עוסקים בה היא first\_fit, המקום הראשון שיהיה פנוי- נתפוס אותו.

שקופית 11: פונקצית init שמאתחלת משתנים גלובלים של גבול הheap.

שקופית 12: free מקבל את הכתובת לתחילת הבלוק מידע. הוא הולך גודל header אחורה, ומעדכן את הבולינאי של הheader לפנוי (1).

**שיעור 4- ב כסלו**

\*כשדיברנו על הheap בו יש זיכרון רציף- הכוונה שהכתובות הלוגיות (בטבלת הכתובות הלוגיות, שהן הכתובות (הרבות יותר מהמציאות) שאנחנו חושבים שיש לנו) הן רצף אחד ארוך, אע"פ שפיזית ייתכן שהזיכרון מפוזר במקומות אחרים. האינדקסים בטבלת הכתובות- הם שמסודרים באופן רציף, וקובץ אינדקסים קרובים ממפים לchunk רציף בזיכרון פיזי.

בהתחלה המערכת לא באמת מרצפת לנו את הheap עד הסוף, אלא מגדירה break שעד אליו נוכל להקצות ואם יגמר לנו המקום- הbreak יוזז ויהיה לנו יותר מקום. אם ננסה לגשת לheap למקומות אסורים, נקבל segmentation foult. כי יש ניסיון לגישה לכתובת שלא ממפה לזיכרון פיזי. לעיתים חריגה ממערך תהיה segmentation fault אם עברנו את הbreak ולעיתים לא אם אנחנו בתוך הגבולות שמערכת ההפעלה הקצתה לנו.

ביוניקס יש לנו 2 פונקציות:

*Brk-* זו פונקציה הפונה למערכת ההפעלה (system call). פונקציה הפונה למערכת ההפעלה היא פונקציה כבדה כי היא תוקעת את מערכת ההפעלה עד שהיא תענה לנו, ולכן מנסים להמנע מsystem calls. פונקציה זו ממקמת Break לפי בקשתנו מראש. מחזירה 0 אם היא הצליחה.

*Sbrk (set break)-* פונקציה זו מזיזה break קיים לפי הצרכים בדיעבד. יש מערכות המאפשרות sbrk שלילי- כלומר כזה שמקטין את הטווח עד הbreak. פונקציה זו מחזירה היכן נמצא הbreak ולכן אם נשלח לה 0- נקבל את מיקום הbreak.

במצגת שקופיות 14-15: מימוש free malloc. בשקופית 14 מאותחלים הגלובלים.

בשקופית 15, free מקבל כתובת לתחילת הבלוק, הוא חוזר אחורה את הheader ומשנה את הביט בו שיגיד שהבלוק פנוי.

בmalloc מקבלים כמה בתים אנחנו צריכים. מוודאים שהמשתנים הגלובלים לתחילת וסוף הheap אותחלו. מגדילים את הזיכרון הנדרש לנו למספר הבתים שהמשתנה ביקש+ גודל header. כעת, כל עוד current location לא הגיע לסוף הheap נמשיך בחיפוש הזיכרון הפנוי. בחיפוש משתמשים בפוינטר של header ופוינטר void\* המתקדם בתים. בשקופית 18 עושים casting לcurrent location מפוינטר void\* לפוינטר לheader כדי שנוכל לקרוא את is available של header. אם הבלוק פנוי- נעבור לבדוק אם גודלו מספיק לנו. אם כן, נעדכן את הheader לתפוס, נקדם את המיקום הנוכחי בheader ונחזיר את המיקום הנוכחי למשתמש.

אם הבלוק לא מתאים- נתקדם לבלוק הבא.

היציאה מהwhile תגרם מכך שלא נמצא בלוק/ הזיכרון נגמר. הmemorylocation משתמש כflag אם נמצא מקום או לא. הוא 0 אם לא נמצא מקום.

אם לא נמצא מקום (שקופית 19), נזיז את הbreak כגודל מה שחסר לנו. המיקום הנוכחי יהיה הlast של מה שהיה מקודם, ונעדכן את last. נתפוס את הזיכרון שהוספנו ונעדכן בheader שלו את הsize. נתקדם עם current את הheader של הבלוק שהוספנו ונחזיר את המיקום הנוכחי למשתנה.

האלגוריתם שראינו הוא פשוט ולכן זמן הריצה שלו מהיר אע"פ שאין אצלו תפיסה חכמה של זיכרון (אפשר למלא בלוק ענק במשתנה קטן...).

בaligned צריכים לקחת שיקולים כמו התחלה בכתובות בחזקות 2 שמתאימות למעבד הקורא words. לכן בלוק עבור 13 בתים- ייתן לנו גודל 16 למשל.

יש מספר דרכים לחיפוש מקום:

**Best fit-** מחפש את המקום הכי טוב להקצאת הזיכרון. יישמר עוד מידע של מה המקום הכי טוב שראינו עד כה ומה רמת הטוב שלו. לכאורה הוא אמור להיות מנצל טוב יותר של הזיכרון- אך יכול לבזבזו בגלל שנשארים חורים קטנים מהבלוקים שאי אפשר לפצלם ולנצלם ולאט לאט הם מתרבים. לוקח לו זמן ארוך. הוא סוג של first fit כי הראשון שטוב להכניס בו- נכניס בו. עוד בעיה בו- היא שכאלה שהוקצו יחד- בהגיון, כנראה נשתמש בהם יחד. למעבד יהיה נוח שהם יהיו יחד בכתובות וירטואליות סמוכות וזה מגביר סיכוי שגם בפיזי הם יושבים קרוב באותו chunk. אולם בשיטה זו הם לא בהכרח סמוכים וזה מקשה על המעבד.

**First fit-** בוחר את המקום הראשון שיש. עלול להיות ניצול לא טוב לזיכרון. אבל מצד שני, יש חלוקה של בלוק ושימוש בשארית מעבר למה שהשתמשנו בו. יגרום לכך שמשתנים קרובים כנראה יוקצו קרוב בזיכרון.

**Next fit-** מחפשים כל פעם בלוק במקום האחרון שהיינו בו ולא מההתחלה וסטטיסית יצא שהגדולים יהיו בהתחלה ונחפש פחות זמן.

שקופיות 29-30: דרכים לדעת איפה המקומות הפנויים ברשימה (במקום לדפוק בדלתות במלון- מחזיקים רשימת חדרים פנויים). מטיילים על הרשימה במקום על הבלוקים.

בשקופית 30 יש רשימה מקושרת שמצביעה למקומות הפנויים בזיכרון ושומרת size. אם איבר מסויים נתפס- הוא יוצא מהרשימה המקושרת ואם הוא מתפנה- הוא יוסף לרשימה.

שדרוג נוסף יכול להעשות ע"י שמירת מערך המקשר לרשימות מקושרות של בלוקים בגדלים שונים.

אפשר גם לשמור bitmap כאשר בלוק תפוס יסומן ב1 ופנוי ב0 ואם גודל הבלוקים קבוע- נדע לאיזה בלוק לעבור.

אפשר לשמור בתים לפי הגדלים שנדרשים בתוכנית הספציפית או מראש לשמור בלוקים בגודל קבוע באיזורים שונים. כך הניהול והחיפוש מהירים בהרבה.

כעת, נרצה לדעת מה זה new ו- delet ועל השוני שלהם מmalloc ו- free.

New (שקופית 19) : מנסים לעשות malloc. אם malloc החזיר כתובת נחזיר אותה למשתמש, ואם malloc יחזיר null יזרק exeption ולכן אין משמעות לבדוק אם חזר מnew NULL כי לא נגיע לשורה הבאה בקוד.

יש אפשרות להכניס לnew פונקצית עזר שכשmalloc נכשל הוא יפעיל אותה לפני זריקה exeption וינסה שוב להפעיל malloc. פונקציה זו תוכל לדאוג לאחד חורים וכדו'. ברגע שכבר לא תהיה משמעות לפונקציה נגיע לexeption שיזרוק אובייקט מסוג bad\_alloc.

בס"ד 12.11.2013

תוספת על שיעור שעבר: קיימות 2 שיטות לשמירת הheader. כחלק מהheap או לשמור אותו במקום חיצוני לזיכרון על הheap באופן הבא: מחוץ להheap יהיה map שמקשר זוג של key: פוינטר למקום בזיכרון ו- value: header. בשיטה השניה, יש פחות סיכון שמישהו ידפוק את הheader ויבלגן את כל הheap.

כשנעשה free נשנה פשוט את הheader בmap.

מערכות real time- מערכת מחשב שיש לה, מעבר לדרישות הפונקציונליות הרגילות, דרישה של עמידה בזמנים. מערכת זמן אמת לא חייבת להגיב מהר, אך היא חייבת להגיב תוך הזמן שהוגדר בדרישות והיא חייבת לבצע זאת בעקביות. לכן אין exception במערכות real time כי אז הגדרת הזמנים לא ידועה מראש. לו היה new המערכות real time, היה למשל new שמחזיר null ולא exception כשהוא נכשל בשביל מערכות real time.

הערה- אם עושים בצורה גלובלית אווור לאודינג לnew delete, משמעות הדבר שאי אפשר יותר להשתמש באלו הדיפולטיבים! מסוכן...

שקופית 25: typedef void(\*new handler\_ptr) () – הגדרה של פוינטר לכל פונקציה שמחזירה void ולא מקבלת פרמטרים.

בשקופית 26 יש פונקציה void ללא פרמטרים. בשורה ראשונה, מוצב בפוינטר לhandler כתובת הפונקציה (זה מה שמקבלים עבור הפרמטר בסוגריים שהוא שם הפוקנציה).

אפשר להחליט שלאחר מספר מסויים של כניסות לhandler לא נכנס אליו יותר. הדרך המעשית לעשות זאת תהיה ע"י השמת NULL הפוינטר לhandler כך שבפעם הבאה שירצו לפנות אליו- הוא כאילו לא קיים וnew תכשל.

Thred- מצב בו מפצלים קוד ככה שכמה אנשים יוכלו לרוץ עליו ולא תעשה עבודה מיותרת כמו הגדרת tokens כמה פעמים.

נרצה כעת להבין מהם new ןdelete.

**Delete:** אם הפוינטר לא NULL, יעשו 2 פעולות: יקרא הDtor וישוחרר הזיכרון. הזיכרון לא ישוחרר בתוך הDTOR!

**New:** קודם קורא לnew(p) שיקצה זיכרון ואז לCTOR. מכך אפשר להבין, שאם לא נמצא זיכרון מספיק פנוי- לא נגיע לCTOR כי יזרק קודם exception.

לעיתים נרצה לשנות new ו- delete כדי להקצות זיכרון בצורה יותר יעילה. הרבה פעמים בתחילת התוכנית נקצה מערך גדול של char כי הם בגודל 1 בית. אח"כ בnew ו-delete אנחנו נהיה אלה שיקבעו איך לסדר את הזיכרון (כי לheap אין לנו גישה, וזה כבד וקשה להתחיל להתעסק בheaders). ככה אנחנו יכלים לשדרג את malloc לפי רצוננו. New ו-delete הכלליים לא יוכלו להשתנות. אלו פונקציות שלא מקבלות פרמטר. מה שנדרוס זה את הnew ו- delete הפנימיים הקוראים לפרמטרים. כך new כללי יקרא לnew שלנו המקבל כתובת בה רוצים להקצות את האובייקט (placement new) ואז לCTOR. Delete כללי יקרא לDTOR ואז לplacement delete שלנו (שמקבל void\* כי כבר הופעל דיסטרקטור ולא זה לא פוינטר לאובייקט).

יש סכנה בדריסת new ו-delete גלובליים כשקוד יועבר בין מתכנתים וכו' ועלול לדפוק את הקוד אחד של השני.

במצגת בתוכנית שמראה מימוש new delete, אם נריץ, נראה שבהקצאת מערך, המחשב יקצה עוד header למערך (עוד int- 4 בתים) שנדרש כדי לדעת כמה פעמים לעשות CTOR וכמה פעמים להפעיל אח"כ DTOR בdelete.

ניתן בעזרת operator overloading לממש operator new בתוך המחלקה, ואז ישתמשו בו רק במקרה שבו יש הקצאה של אובייקט מסוג המחלקה. זה יראה כך:

*void\* operator new(size\_t);*

*void operator delete(void\*);*

הheader יקטן כי הsize קבוע, ונוכל לנצל זיכרון ללא חורים ולא נצטרך לחפש מקום פנוי אלא נלך למקום האחרון שהתפנה וזהו.

במצגת קוד לדוגמא: החל משקופית 11. שקופית 12: מוגדר מערך סטטי בגודל היכול לאכסן 100 אובייקטים. מערך מקביל יגיד על כל בלוק אם הוא פנוי או לא (בהתחלה מאותחל ב0. הכל פנוי).

אח"כ מוגדרים CTOR וDTOR של האובייקט שסתם מבצעים הדפסות.

הnew החדש שיוגדר (שקופית 13), ירוץ על הmap בו יש מערך עם אותו מספר תאים אך כל תא בגודל בית אחד ולא בגודל האובייקט כמו הזיכרון עצמו שמוקצה. אם יש מקום פנוי- נעדכנו לתפוס, ונחזיר פוינטר למקום במרחק אינדקס המערך\*גודל אובייקט= המקום במערך אליו הגענו.

אם כל הזיכרון תפוס, נוציא פלט מתאים ונחזיר NULL. המשתמש בפועל יקבל פוינטר לstack אע"פ שהוא עשה new.

בdelete, נגיע לאינדקס בmap של המקום המשוחרר ע"י חלוקת הכתובת בsizeof של האובייקט ואז נעדכן את האינדקס המתאים ב0 לומר שהוא פנוי.

**מצגת 3- ניהול זיכרון**

קונסטרקור ימלא משתנים לפי הסדר שהם הוגדרו במחלקה ולא לפי הסדר שישלחו ברשימת אתחול ולכן יש משמעות לסדר בו מגדירים משתנים.

New[]- בהתחלה קורא להקצאת זיכרון עם n\*sizeof ואח"כ קריאה לקונסטרקטור n פעמים עם הכתובת בקפיצות sizeof כל פעם. הגודל שיוקצה יהיה הגודל+ 4 בתים לint כדי לעזור אח"כ לfree לדעת כמה אובייקטים הוא צריך למחוק. כלומר יש header גדול כמו בכל הקצאה, ועוד int שנוסף במערכים כדי לדעת כמה איברים לשחרר אח"כ. לכן, שימוש בdelete[] כשצריך delete, יעשה בלאגן כי הקומפיילר הולך 4 בתים אחורה לחפש גודל ויקבל שם זבל כלשהוא.

שקופית 20: הסימון :: מסמל לפנות למשתנה גלובלי.

כמו ראינו על header, גם במערך יש 2 אפשרויות איפה לשמור את גודל מערך. או לפני המערך, או בכתובת חיצונית שממפה כתובת מערך לגודל שלו.

שקופית 28: יש הקצאת מערך char בגודל האינטג'ר+ גודל המערך. אח"כ מגדירים פוינטר למערך כזה. יש הכנסה של int ל4 בתים ראשונים שהוקצו.

אח"כ רוצים רק להשתמש בconstructor שמקבל כתובת- אבל אם נקרא לCTOR כזה דרך new יוקצה זיכרון. לכן C++ הגדירה placement new – new שמקבל כתובת ומה שהוא עושה זה רק לקרוא לCTOR על שטח זיכרון קיים ולא להקצות זיכרון חדש. זה מה שנקרא ב28 באמצע. אח"כ יש catch למקרה שיהיו בעיות.

ההפכך מplacement new, הוא רק לרוקן את תוכן זיכרון אך לא לעשות free ולכן המצב היחיד בו אפשר לקרוא לDTOR כמו שרואים בcatch, זה כהיפך לplacement new.

שקופית 29: יש קריאה לDTOR בלולאות ולאחר מכן קריאה כללית לdelete על כל המערך.

שקופית 32: יש מפה שממפה גדלים לכתובות בזיכרון.

אם נקרא לdelete רגיל על משהו שהוקצה עם [], רק האיבר הראשון ישוחרר.

אם נקרא לdelete [] על משהו רגיל שהוקצה, נלך אחורה לקרוא את הint, שם נגלה זבל ולך תדע איזה לולאות DTOR לא קשורות יקראו...

Free- מדבר, כמו malloc, ברמת הבלוקים. אם הגודל צמוד למערך, יש בבלוק header ואז int של גודל מערך ואז את המערך עצמו. אם נלך אחורה header מנקודת תחילת המערך, נגיע 4 בתים בהזחה קדימה ממה שרצינו והheader יקרא בבלאגן.

אם המימוש של הגודל הוא בצורה שהגודל נמצא חיצוני מהבלוק במפת מיפוי, new[] עם delete לא נלך לטבלת הגודל, הקומפיילר יקרא רק לDTOR של הראשון וילך את הheader אחורה וישחרר את כל הבלוק ולכן אם למשתנים הבאים בבלוק יש הקצאות- יהיו דליפות כי DTOR שלהן לא נקרא.

אם יהיה new וdelete[]- הקומפיילר ינסה ללכת לכניסה בטבלה ולא יצליח למצוא מקום ובהתאם למה שבחר המתכנת יוחלט מה יקרה.

השיטה של map בטוחה יותר, אך דורשת יותר זיכרון, וזמן ניהול של הmap.

שקופית 33- מראה איך נראה delete[].

New שלישי שיש הוא כה המקבל אובייקט nothrow. New כזה, הוא new שמחזיר NULL ולא exception אם הוא נכשל ולכן יש חובה עלינו לבדוק אם חזר NULL.

בסה"כ ראינו 3 new:

1. רגיל
2. מקבל כתובת ומפעיל רק CTOR
3. Nothrow שמחזיר NULL בבעיות.

**מצגת נושא 4 garbage collection**

Gerbage מוגדר כאובייקט שנמצא על הheap ואין עליו הצבעה מהstack/ מהDATA (ממקום מנוהל).

לפעמים יש דברים שמבחינתנו הם garbage אע"פ שיש עליהם הצבעה מהstack. לכן, ראשית נצטרך להגדיר מהו garbage.

בשבוע הבא נראה 3 שיטות כיצד להגדיר ניקיונות (כמו למשל reference count שדואג שלכל אובייקט בheap יש מצביע על הstack) ועוד 2 אלגוריתמים שנלמד שבהם משתמשת JAVA למשל. נראה כיצד יכולות להיות דליפות בג'אוה.

שבוע הבא:

יתרונות garbage collection:

* הופך את הקוד ליותר אמין שלא נעוף ולא יהיו dangling pointers.
* המערכת עושה לנו את העבודה.
* פחות זמן התעסקות בדיבאגינג.

חסרונות:

* הרבה פעמים התוכנית צריכה לעצור בשביל הניקיון (הרבה פעמים ננצל את המומנטום לסידור הזיכרון)
* שימוש בזכרון מעבד.

שיטות להפעלת GC:

*1. התיחסות עם reference count שסופר כמה מסתכלים עלי.*

כמו שעשינו שיעור שעבר. מי שאין גישה אליו מSTACK או מהHEAP- יימחק והcount שלו 0= ימחק כי לא מסתכלים עליו. המימוש נעשה ע"י מחלקה עוטפת שבה יש פוינטר למחלקה ומשתנה סטטי שסופר כמה מצביעים יש. הsmart pointer ידאג בדיסטרקטור למחוק מי שלא מצביעים עליו, ונממש לו אופרטורים כמו לפוינטר רגיל.

נרצה לדאוג כמובן שהsmart pointer לא יוקצה על הheap.

יש חריגות מהרעיון כשיש בheap יש מעגלים פנימיים כך שהקאונטר סופר אותם ולמרות שאין גישה מהstack משתנים לא ימחקו.

יתרונות refence count:

1. לא עוצר את התוכנית ויש מחיקה מיידית ברגע שצריך.
2. קל למימוש

חסרונות:

1. עבור אובייקטים קטנים, ההגדלה בזיכרון לעוד פוינטר+ integer משמעותית. לאובייקטים גדולים זה ישתלם.
2. לא מזוהים מעגלים ובמעגלים תהיה דליפת זיכרון.
3. על כל העברה של אובייקט יש כל הזמן התייחסות לcounter של האובייקט כרעש רקע.
4. לא מטפל בחורים בזיכרון כי אין הסתכלות על הheap בכללותו.

נראה כעת 2 שיטות אחרות, tracing, המסתכלות במבט רחב יותר את הזיכרון כולו.

*2. mark and sweep:*

נסתכל על האובייקטים שבSTACK ובHEAP כשורשי עץ ונתחיל לטייל מהם בDFS תוך כדי צביעה מי שיש קישור אליו מהשורש. אח"כ נעבור על הHEAP, וכל מי שלא צבוע- נסמן בHEADER שלו שהוא פנוי. כך לא נפספס מעגלים. תוך כדי ריצה על הHEAP נמחק צבע ממי שצבוע כהכנה לפעם הבאה שgarbage cokkection יעבור פה.

יתרונות:

* לא מפספסים כלום ונפתרה בעיית המעגלים.
* אין משחק עם פוינטרים אלא עבודה סדרתית במחיקה.
* נדרש רק ביט ולא המון זיכרון לכל אובייקט כמו בreference COUNT/.
* פשוט

חסרונות:

* התוכנית צריכה להעצר כי אם נוצר אובייקט אחרי הMARK הוא לא יסומן וימחק. אפשר לפתור את זה בעזרת FLAG שיאמר על אובייקט שהוא נוצר בזמן שאחרי MARK ואז לא נמחק אותו אח"כ כשננקה את הHEAP.
* זמן ריצה ארוך לסמן את כולם, ולעבור כל פעם על כל הHEAP. צריך לייעל sweep.
* אין טיפול בחורים ומייצרים עוד חורים.

(\* כל הנתונים האלו פנוי/תפוס, צבוע/לא צבוע וכו' יכולים להעשות על CHAR אחד כשכל ביט שלו מייצג מידע אחר. משתנה ביטי כזה הוא MARK).

*3. stop and copy*

מתלכתחילה המשתמש יקבל רק חצי מהHEAP לשימוש . במידה וNEW נכשלת, הHANDLER שיתבצע הוא האלגוריתם הזה, שייקח את החצי המבולגן שלנו המלא, וימלא איתו בצורה מסודרת את החצי הריק. במקום לסמן כמו באלגוריתם הקודם- הפעולה שתתבצע היא שמירה בחצי הריק של הHEAP. כך כל פעם יהיה שימוש בחצי אחר של הHEAP. כמובן שיחד עם פעולת ההעתקה יש לעדכן פוניטרים למיקום החדש בזיכרון של המשתנים.

אפשר גם לשמור את המידע עד איפה מולא החצי בHEAP שאליו העברנו את המשתנים ובפעם הבאה שנרצה להקצות זיכרון- נדע ישר היכן יש מקום פנוי.

יתרון:

יש רק מעבר אחד על הסימון

אין מעבר על כל הHEAP אלא עבודה פרופורציונאלית למספר אובייקטים חיים.

מסדר יפה את הזיכרון

פותר בעיית מעגלים.

חסרון:

עלות של חצי HEAP כי כל פעם עובדים רק על חצי מהHEAP.

התוכנית חייבת לעצור בכל הפעלה של האלגוריתם.

הפתרון הזה לא יעיל לC++ כי זה יצור המון בלאגן בשפה שיש בה עבודה עם כתובות בפוינטרים ואנחנו משתנים את התוכן של מה שהפוינטר מצביע עליו.

*4. generation GC*

שיטה זו רוצה לשפר את MARK. היא עובדת עם סטטיסטיקות שאומרות שאובייקטים צעירים עם יותר סיכוי למות מאובייקטים זקנים. במקום לעבור ישר על כולם למרקר אותם, מנסים בהתחלה להסתפק בבדיקה רק של הצעירים ורק אם לא מצליחים נמשיך בהדרגה לסמן זקנים יותר. זקנה תוגדר לפי מספר מחזורי הGC שאובייקט שרד.

נגדיר בHEAP תינוקיה שעליה נפעיל MARK & SWEEP ומי שישרוד אותו- יועבר לאיזור בוגר יותר. אם עדיין אין מקום, בכניסה הבאה לHANDLER נעבור לחלק של בוגרים יותר ונפעיל עליו MARK AND SWEEP. מי ששרד 2 מחזורי GC- סטטיסטית ישרוד לנצח ואם נכשל שוב לא ננסה שוב להכנס. כלומר יוגדרו מספר איזורים לגילאים מסוימים, ובשלב מסוים יווצר אזור בני אלמוות שעליו כבר לא יופעל GC.

כך עובדת JAVA. זו למעשה מין תת שיטה של stop & copy כשהחלוקה לאן להעתיק נעשית יותר בחכמה.

**חנוכה**

נרצה למיין אובייקטים כדי לא לעבוד על כולם בכל הפעלת GC. בgeneration, החלוקה נעשתה לפי משך הזמן שהאובייקטים קיימים. בJAVA יש פתרון אחר ע"י קבוצות אובייקטים הנוצרות באיזורים שונים בHEAP, וGC יופעל על בלוק אובייקטים בHEAP שיש לו הכי פחות reference count ולכן הכי סביר שיש שם הרבה מי למחוק.

בניגוד לc++ בה השליטה על הקצאות היתה עלינו והיו בעיות של אתחול ומחיקה (שפתרנו עם CTOR ופתרנו חלקית עם DTOR). את רעיון הCTOR לקחה JAVA ועם המחיקה יש התמודדות עם GC. אלא, שיש בעייתיות שכן DTOR מטפל בכל מיני פעולות שיש לבצע בעת מחיקת אובייקט (סגירת קבצים וכדו'). לכן יש פונקצית finnalize שבעת הפעלת GC על אובייקט מסומן תקרא. צריך לקחת בחשבון שלעיתים לא נקרא בכלל GC ולכן לא בהכרח תקרא finallize.

**finnalize**

יש את הJVM היודע לשמור root, וסימון למי מהאובייקטים יש finnalize. לאובייקטים אלו שיש להם finallize, נשמור מחסנית עם פוינטרים אליהם. יש עוד תור שבתוכו נכניס פוינטרים לאובייקטים שרוצים למחוק בהרצת GC.

בעת הרצת GC,

קודם נעבור על כולם ונמחק את מי שאין לו finnalize וצריך למחוק, ומי שיש לו finnalize , יופעל finnalize בthred אחרי שGC יסתיים, והוא יוסר מתור הfinnalize . בהרצת GC הבאה, אובייקטים אלו ישוחררו. היתרון בדרך זו היא המהירות בה יופעל GC שרק מוחק, אולם צריך להזהר שכן אובייקטים עם finallize ימחקו רק אחרי 2 מחזורי GC.

לא ידוע מה סדר הקריאה לfinnalize ולכן צריך להזהר מתלות בין finnaliz שונים. כמו"כ צריך לקחת בחשבון שייתכן שהוא לא יקרא אם לא יופעל GC.

System.gc()- פקודה המזמינה GC (לא ברור אם זו המלצה או שבהכרח זה יקרה).

**Weak reference:**

לפעמים רוצים לעשות reference לאובייקט שלא יעקב ממחיקה שלו בGC. דוגמאות:

* reference מהcatch לאובייקט בRAM
* Reference לאובקטיים במערך שעליהם רוצים לעבוד לחישוב מסוים. כשיופעל GC על המערך, לא כולו ימחק כי על חלקו יש reference. בגלל שמישהו הסתכל עלי- אני לא יכול להמחק.

בכדי לפתור את הבעיה, יש את רעיון הweak reference שהוא reference שGC מתעלם ממנו וימחק אובייקט שיש עליו רק הצבעת weak. הבעיה נפתרת ע"י רמות של הצבעות.

(במצגת בשקופית 29 שמדגימה שימוש, 2 שורות מהסוף- פקודת sleep שנותנת לgc זמן לרוץ).

**Memory leaks in JAVA**

לכאורה פתרנו בJAVA דליפות זיכרון, אך עדין יש דליפות, בעיקר לוגיות.

יש 2 מונחים:

lost object- אלה שאין אליהם refernce

useless object- כאלה שיש reference אך אין צורך בהם. אלו אובייקטים loyter (בטלנים) אי אפשר לשחררם כי השחרור בprivate או ששכחנו לעשות להם weak refence למשל ויש להם refence למרות שאין צורך בהם. כיצד תפתר הבעיה?

תחום זה נמצא במחקר ויש תוכנות חיצוניות הרצות על התוכנה ובודקות מי הבטלנים.

תוכנה חיצונית כזו, יכולה להריץ את התוכנה המקורית ולאסוף סטטיסטיקות על מי לא עובדים. התוכנות יכולות למצוא צווארי בקבוק- פונקציה שקוראים לה הרבה יותר מפונקציות אחרות והיא תוקעת את התוכנית בתלות של כולם בה, וע"י פיצול שלה, התוכנה תרוץ ביתר קלות.

התוכנה תוציא לנו פלט שנוכל דרכו להבין איפה שכחנו לבטל refence וכדו'.

**הנדסת תוכנה**

הרעיון בהנדסת תוכנה הוא למצוא שיטתיות לתהליך מקבלת הזמנה מלקוח ועד הבאת התוכנה ללקוח (הן אם מדובר בפרוייקט שלם והן אם זו הרחבה של פרוייקט קיים). יש design pattern לשלבים שונים בתהליך וצריך לזכור שבסופו של דבר המטרה היא רצון הלקוח. לעיתים נעדיף לקנות חלק מהפתרון ולא לבנות חלק זה מבחינת עלות (שכן מתכנת עולה כ- 1200 דולר ליום).

כמו"כ, צריך לשים לב על התקשורת בין צוותי הפרוייקט, ועל עלות מול מתחרים.

פרוייקט מורכב מלקוח (מי שמשלם), משתמש, מפתחים ומנהלים. בין כולם יש ניגוד אינטרסים שצריך לאזן.

מערכת מורכבת מתוכנה וחומרה (החומרה היא HW ןFW שזו תוכנה הצרובה על התוכנה). CSCI זה רכיב תוכנה.

חלק ניכר מהעבודה שפרוייקט מושקע בתחזוקה לאחר שהלקוח קיבל את התוכנה, שכן רק אז נעשות בדיקות במלואן כמו שצריך בתנאי אמת.

**מתודולוגיות פיתוח:**

מתודולוגיות קוויות: תכנן תקן. כותבים קוד ומתקנים אותו כל פעם בהתאם לבאגים. זו מתודולוגיה מהירה המתאימה לפרוייקטים קטנים אך יש בדרך זו בזבזנות וקשה להעריך כמה זמן עוד יקח.

מודל מפל המים: מודל המתאים לפרוייקטים גדולים. במודל זה, יש שלבים של דרישות, תכנון, מימוש וכו' ועד ששלב אחד לא מושלם- לא נעבור לשלב הבא ולא חוזרים אחורה בשלבים. אם יש בעיה- נחזור לשלב הבעייתי (למשל דרישות משתנות) ונתחיל לרדת מחדש במפל השלבים. זה מודל סדרתי. החסרון היא שקשה לחשוב על הכל מראש והמתכנתים מבוזבזים כשלוקח כ"כ הרבה זמן עד שנגיע לקידוד.

מודל ספיראלי: עובדים בשיטת מפל המים בכל פעם על קטע מהפרוייקט שהולך ומתהווה מחיבור עוד ועוד קטעים כאלו. יש בספיראלה יתרון שמבינים אם לא הבנו משהו נכון קרוב לבעיה שנוצרה כי ההתקדמות מובנת ואיטית בהדרגה. החסרון הוא באובדן שליטה כי אין עבודה בכללי על כל הדרישות וכדו' ואפשר לאבד את הצפון.

מודלים זריזים- Agile: מחלקים את הצוות לצוותים קטנים עם משימות קטנות לזמן קצר כך שתוך זמן קצר יש מוצר עובד משילוב העבודה של הצוותים הקטנים. בכל בוקר יש ישיבה בה כל אחד מדבר דקה על מה עשה ומה יעשה היום. הישיבה מתבצעת בעמידה כדי לתקתק עניינים. יש טבלה לכל אחד של מה עשה ומה הלוז המדוייק שלו. זה מה שהולך היום- מעקב צמוד ותקתוק זמנים.

בס"ד י"ד טבת תשע"ד

כשחברה מקבלת פרוייקט היא מתמחרת אותו לפי שנות אדם- כמות הזמן שייקח למתכנת לעשותו (עובד חדש יתומחר פי 2 זמן), ומחיר שנת אדם הוא המשתנה בין החברות השונות.

מתודולוגיית scrum עליה דיברנו בשבוע שעבר, היא מתודולוגיה שבה כל הצוות עובד יחד על משימות קצרות בטווח של שבועיים.

בעיה נפוצה בתעשיית ההיי טק, היא בריכוז מומחיות אצל אדם מסוים, ובאמצעות scrum כשהמידע מפוזר, אין תלות בעובדים. אעפ"כ כדאי שתהיה הנישה שלך שלא יפטרו אותך. נקודה זו היא גם חסרון בscrum כי נותנים לעובדים משימות בתחום שהם פחות אוהבים.

מתודולגיית פיתוח מונחה בדיקות (Extreme Programming)- חושבים מה הלקוח ירצה שנממש ובהתאם לכך בונים את התוכנה. פרוייקטים במתודולוגיה זו קיצוניים כשכל הזמן התכנות מלווה בבדיקות. התכנות בדרך זו נעשה בזוגות כשאחד עובד והשני מנווט אותו בביקורות ברקע.

\*צוותי RNP- צוותים שעסוקים בגנריות של קוד כדי שיוכל לשמש בעתיד לצרכים נוספים. לצורך מימונו, מלכתחילה מעגלים קצת למעלה את הזמן שייקח לממש את הפקרוייקט.

**ארכיטקטורה:**

כריסטופר אלכסנדר היה אדריכל ואיש מדעי המחשב. הוא תכנן מלון ובפעם השניה שביקשו ממנו לתכנן מלון, הוא הכין מין design pattern לבניית מלון וכך המשיך את רעיון הdesign patterns לתחום התוכנה, כשיש פתרונות ידועים לבעיות שכיחות בdesign של מערכת.

הכוח הרציני בdesign pattern הוא בשפה שהוא יוצר בין המתכנתים והre-use שלו.

Implementation patterns- תבניות המדברות על איך לכתוב יישומית קוד נכון. בהקפדה עליהן, הקוד ברור גם בלי הערות כי השמות משמעותיים וכו'. דוגמאות לתבניות:

1. שמות משמעותיים.

2. חלוקה נכונה של פונקציה גדולה לפונקציות קטנות.

3. פיצול חישובים גדולים לתת חישובים שיחד יוצרים נוסחה סופית.

בארכיטקטורה של בניין לוקחים בחשבון תנאי שטח, תשתיות, צרכים ותקציב ומנסים לחבר את כולם לתכנון התוצר המוגמר. בארכיטקטורת תוכנה בדומה לכך, יש תכנות של מבנה התוכנה בהתאם לשיקולי ביצועים, תפוקה, זמינות, סלחנות לבאגים, גמישות לשינויים ועוד.

דוגמא לארכיטקטורה: OOP.

כשיש פרוייקט מתחילים מפיצול שלו לתתי בעיות עד שמגיעים לרמת יישום שכבר תשתנה בהתאם לסביבת העבודה. הארכיטקטורה שנבחר בסופו של יום תלויה במספר המחשבים העובדים ועוד.

*דוגמאות לארכיטקטורות:*

פרצדורלי- קוראים לפונקציה ומקבלים תשובה.

Virtual machine-

Data Flow-

Event based- כך עובדת windows. כשנעשה שינוי, נשלח event ומערכת ההפעלה צריכה להבין מי אמור לטפל בשינוי (למשל לחיצת כפתור) שנעשתה.

Blackboard- נניח שנרצה לממש סימולציית משחק כדורגל ונעבוד עם OOP- זה לא יעבוד כי לא הגיוני שנרצה עבור כל שחקן בכל רגע get למיקום הכדור והשחקנים האחרים והסימולציה תקרוס. Blackboard היא ארכיטקטורה בה יש לוח מרכזי המכיל את המידע שדרוש לכולם לדעת לפי id מזהה ליישויות. יישות לא תהיה מחלקה, אלא id בטבלת המיקומים, id בטבלת האביזרים וכו' ומתודות שיחד הם למעשה מיישמים יישות. נדרשת משמעת מהמתכנתים לא לעדכן מידע שלא באחריותם כי אין למעשה הגנה של private כמו בOOP.

הבעיה בארכיטקטורה זו היא בסנכרון כשיישות אחת יכולה להשתנות ואנחנו קוראים מידע שכבר לא רלוונטי. למשל כדור כבר עף אבל אני עם המידע הקודם שקראתי עליו.

בשלב הארכיטקטורה עוד אין design אלא רק מחשבה ברמה כללית יותר, ללא הבנת לעומק של המערכת, כיצד היא תבנה. מחזיקים controller שהוא דואג לתיאום עם כולם. המערכת רצה בכמות מסוימת של Hz המגדיר כל כמה זמן כולם מתעדכנים. כדי שראשונים ואחרונים שרצו באותו סבב לקבל מידע לא יקבלו מידע שונה, נשמש בשיטה הדומה לstop and copy כך שבסבב אחד כולם מקבלים את אותו מידע.

יש תוכנות בהן יש כמה סבבים התלויים זה בזה ולכן מנסים להבין מה הקו מגמה של אחד הסבבים ובהתאם לכך עודכנו סבבים אחרים ואם קו המגמה לא היה נכון- נעשה תיקון לבעיה שנוצרה.

Client server- זו ארכיטקטורה של יחסי לקוח הצורך מידע ושרת המעניק לו את המידע. דוגמא: יש מערכת שאמורה רק להציג נתונים, כשמישהו אחר ידאג לעיבוד של הנתונים (למשל מערכת הצגה של נתונים בבנק). המערכת היא רק הקלט והפלט והארכיטקטורה אמורה לתאום למטרה זו. החסרון בארכיטקטורה זו היא בתלות של כולם ברכיב אחד. לכן צריך להחליט איך מחלקים את הרכיב שלא כולם תלויים באחד. פתרון אפשרי הוא ע"י peer to peer שזו דרך תכנון בה כולם שרתים וכולם לקוחות. כולם יכולים להתחבר לכולם במין ספגטי וכך אם אחד לא זמין- אין פגיעה טוטאלית בכולם. צריך לוודא בשיטה זו שאין מישהו אחד איטי התוקע את כולם (למשל הGUI איטי וכולם נתקעים בגללו) והניהול של מי ירוץ איפה עולה לנו.

ארכיטקטורת שכבות- מחלקים לשכבה שמחזיקה תצוגה, שכבה של לוגיקה ושכבה של מסד נתונים.

SOA- שיטה בה מנסים להעזר באחרים וכמה שפחות להמציא את הגלגל מחדש (למשל נקנה GUI שחברה אחרת כבר פתחה וכדו).

כ"א טבת תשע"ד

בשבוע שעבר דיברנו על ארכיטקטורת פרוייקט שמדברת על החומרה שתשתתף בו (מחשב/ כמה מחשבים/ פלאפון וכו'), השפה בה נשתמש וכו'.

דיברנו על שיקולים שונים כמו: תלות בקריסת מחשב אחד, עד כמה הביצועים חשובים (משפיע על חלוקה לprossess כך שדברים יכולים לרוץ במקביל בmulty tasks וזה דורש תיאום וכדו') ובכללי על איך להריץ קוד כתוב. זו שאלת הארכיטקטורה.

ראינו client server בו כולם דרך router משותף פונים לשרת (ולכן מתכנת של שרת לא יכול להרשות לעצמו לקרוס- כולם תלויים בו). פגיעה בrouter תפגע בכולם כי כולם תלויים בו.

יש מערכות שלא סובלות את התלות בrouter לכולם ולכן עוברים לpeer to peer בו כולם מחזיקים id אחד של השני וכולם גם client וגם server.

יתרונות וחסרונות peer to peer:

יתרונות:

* פחות תלות זה בזה כי כוח השרת מתחלק בין כולם
* חוסר צורך להשקיע באדמניסטרציה.
* קל להגדיל את המערכת כי לא צריך בשביל זה יכולת כניסה מטורפת של router אחד.

החסרון:

* מחשב חזק המחובר למחשב חלש לא יגיע למקסימום הביצועים שלו בגלל התלות שלו בשני.

**N tier architecture-**ניתן לחלק client sever לשכבות של בסיס נתונים, שכבת לוגיקה ושכבת GUI, שכבת התצוגה, כשביניהם יש interface. כל הרעיון של ארכיטקטורה זו הוא בהפרדה בין תצוגה וניהול מידע. כל צוות העובד על שכבה, נעזר רק בinterface של השכבה מתחתיו, ודואג לממש interface למי שמעליו. לGUI חייבים שלא תהיה תלות בDB. היתרון בכך הוא במידול.

(בד"כ העבודה בGUI נעשית במערכת ההפעלה MAC). יש עבודה כמו pipe כשזרימת המידע בין השכבות חד כיווני. זה נקרא pipe and filter. זרימה לכיוון אחד תוך כדי סינון מידע בדרך.

**Plug in:** כשלמדנו על design pattern של observer, כדי להתעדכן ולהשתתף בתוכנה, אובייקט היה צריך להכנס לרשימה ונדרש linking מחדש. ע"י plug in, רכיבים בתוכנה מציגים פרוטוקול כיצד ניתן להתחבר ולתקשר איתם ואז לא נדרש לינקינג מחדש. יש עבודה של חיבור כל פעם למי שרוצים לרוץ איתו.

**MVC-** הרעיון מאוד דומה לשכבות עליהן דיברנו קודם ב N tier architecture. יש שכבת נתונים, תצוגה, וcpntroller המקשר ביניהם ומטפל בקלט של המשתמש. View עוסק בקלט ופלט ודרך controller מבקש מידע מmodel, שמחזיר מידע לview. ההבדל משיטת השכבות הוא שבשיטת השכבות הזרימה היא ליניארית כשכל שכבה מתעדכנת בזו שמתחתיה. כל התקשורת נעשית דרך השכבה האמצעית. הניגוד לכך, בMVC, יש ארכיטקטורה משולשת. View מעדכן controller שמעדכן model המחזיר מידע לview.

**SOA-** היום חברות כל הזמן קונות זו את זו ומתמזגות וזה דורש אופציה של עבודה מראש שתאפשר את זה. עובדים כמו מין דפי זהב כשחברה מציגה שירות שהיא יודעת לתת ע"י הצגה לפי פרוטוקול עולמי של XML והתקשורת בין חברות עוברת להיות דרך קבצי XML.(אוספים את UI שצריכים מקובץ עם נתונים).

זו דרך שיש בה הרבה reuse, אולם בניגוד לOOP יש כאן ממש פגיעה בכימוסים.

מצגת 7:

נרצה לעסוק בדוגמאות מה לא לעשות בעולם תכנות ANTI PATTERNS.

דוגמאות יש בשלב הפיתוח, הארכיטקטורה והניהול.

**Anti pattern בשלב הפיתוח:**

1. **blob:** זוהי מחלקה ענקית (בד"כ נקראת manager) שמכסה כמעט את התוכנית ויחד איתה יש רק עוד מעט מחלקות קטנות. זה לא באמת OOP. המלצה להמנעות- לא לקרוא למחלקה manager כך שנרגיש שהיא יכולה להכיל הכל. בד"כ הבעיה נגרמת ממתכנתים שלא יודעים לתכנת בOOP, או מחוסר הגדרות מה הדרישות כך שכותבים ברצף בלי חלוקה/ הגדירו ואין code review. זה יכול גם להגרם מהוספה בדרישות הגורמת להוספה למחלקות קיימות במקום לשנות את מבנה התוכנית כולה, או מהגדרות גרועות בלי חשיבה מחודשת של המתכנת איך לחלק את המחלקות.

כדי לפתור את הבעיה:

* שלב ראשון: מנסים לחלק המתודות במחלקה הגדולה למחלקות שקילות.
* שלב שני: ננסה להכניס מחלקות שקילות אלה למחלקות קטנות קיימות אחרות, אם זה נכון לוגית.
* שלב שלישי: חושבים מחדש על הקשרים בין המחלקות

1. **poltergeist-** הבעיה ההפוכה כשיש המון מחלקות כשכל אחת כמעט לא עושה כלום וקשה לעקוב. הפתרון יהיה באיגוד של מחלקות קטנות אם הן קשורות לאותו נושא או להכניסן למחלקה גדולה כמחלקות פנימיות.
2. **ambiguous viewpoint**- כשכותבים קטע קוד, צריך לחשוב למי כותבים אותו. אם כותבים למשל קוד לטלפון- אם הוא למשתמש הוא רוצה נוחות ומהירות, החברה רוצה billing וניהול חיובי לקוחות הכי טוב.
3. **cut and paste**.
4. **fetish-** אהבת יתר לdesogn pattern. חפירות יתר.

**Anti pattern בארכיטקטורה:**

1. ניסיון להמציא את הגלגל מחדש במקום להעזר בדברים קיימים.
2. Vendor lock in- הארכיטקטורה קושרת את התוכנה למוצר מסוים. יש תלות במוצר/ רכיב שאח"כ תקועים איתו.
3. Stovepipe system- במקום לבנות ארכיטקטורה שוכל לעזור גם לאחרים, כל אחד עושה ארכיטקטורה לעצמו.

**Anti pattern של ניהול- אלו צורות של איך לא להתנהל:**

1. Analysis Paralysis- שאיפת יתר לשלמות בניתוח שתעלה אח"כ באי ההתפשרות בפרוייקט.
2. corncob- אנשים סגורים ומעצבנים בחברה שאין איך לגשת עליהם.
3. אנשים שמשתגעים רגע לפני סיום פרוייקט כי הם לא מסוגלים להחזיק את עצמם ברגע האחרון, רגע לפני החופש.
4. smoke and mirror- התחייבות על מטלות לפני שבררתי איך אני מבצעת אותם. אח"כ במימוש אפשר לגלות שאין לנו מושג איך אנחנו נעשה את מה שהתחייבנו.

Refactoring- זהו שלב בקוד בו מחליטים לכתוב אותו מחדש/ לבצע בו שינוי. זה יכול להתבצע בהתחלה כשמגששים לראות איך לעבוד. מצב קריטי יותר הוא כשמגלים את זה לקראת סיום ויש כללים איך לא ליצור בלאגן. למשל הכלל הראשון יהיה לשמור מה שהיה לפני השינוי. נדרשת הסתכלות חיצונית על הקוד. בד"כ מכניסים זמן refactoring בעלות פרוייקט. זה מעבר על הקוד והפיכתו ליותר קריא.

דוגמאות נוספות מה לא לעשות- במצגת (לא למבחן).

למבחן: יש שאלות סעיף תיאורטיות, התמקדות בעיקר בהקצאות זיכרון, אלגוריתם חדש של הקצאה שצריך לנתח מבחינת החורים שהוא יוצר, מהירותו וכו' לעומת אלגוריתמים אחרים שלמדנו (כנ"ל בGC חדש). נושאים שוליים כמו הנדסת תוכנה אולי יהיו סעיף. השוואות בין ארכיטקטורות/ צורך להחליט באיזו ארכיטקטורה נשתמש וכו'.

איגור נותן שאלה שתיים בJAVA.

*תהיה שאלה במבחן אבל לא צריך לזכור בע"פ אלא לדעת כבר מה זה אומר על בסיס דף נוסחאות (גם בראיונות עבודה שואלים את זה):*

Software development principle: אלו 5 עקרונות שמתוכם הומצא OOP. כללים אלו אמורים לעזור לנו להבין איך נכון לחלק את המחלקות שלנו. ראשי התיבות של העקרונות הם SOLID:

1. SRP, **S**ingle **R**esponsibility **P**ronciple- מחלקה צריכה שתהיה לה אחריות אחת בלבד. אם מחלקה אחראית על כמה דברים- נפצלה לכמה מחלקות. במצגת יש דוגמא לבעיה כזו במחלקה שגם אחראית לראות למי לשלוח מייל ומי הלקוחות שלא קנו לאחרונה, וגם אחראית לשליחת מייל. לכן מחלקים את המחלקה ל2 מחלקות שאחת אחראית על ייבוא הלקוחות שאליהם ישלח המייל, ומחלקה נוספת שאחראית על שליחת מיילים למערך לקוחות שנשלח אליה (כי לא באחריותה לייצר מחלקות שיטפלו בטיפולים פנימיים כמו יצירת מערך לקוחות), שהכי נכון שהוא ישלח אליה כאינטרפייס כך שתהיה הפרדת רשויות ונמנע תלות בין מחלקות. הכוח הגדול באינטרפייס הוא שאם פונקציה במחלקה שלי עובדת עם אינטרפייס של פרמטר שהיא תקבל, נוכל לקמפל ולעבור לינקינג עם הפונקציות שנקרא להן מהאינטרפייס ולא נהיה תלויים במימוש של מי שישלח לנו את האובייקט. כך מתכנתים עובדים ע"י סיכום ביניהם מה האינטרפייס של כל אחד ותאפשרת עבודה במקביל. החינוך הוא לחשוב על אינטרפייס ולא על מימוש. פירוט יתרונות- במצגת. נדרשת חשיבה מונחת ממשקים.
2. **O**pen **c**losed **p**rinciple,OCP- הקוד צריך להיות פתוח להרחבות וסגור לשינויים. כלומר, אם רוצים להרחיב את התוכנית נרצה שזה יתבצע ע"י הוספת קוד קיים ועבודה איתו בלי לשנותו. דוגמא לכך בOOP היא בירושה. כלל זה היה הבסיס להמצאת רעיון האינטרפייס. האות I בתחילת מחלקה מייצגת את העובדה שהמחלקה היא אינטרפייס.
3. LSP, **l**iskov **s**ubstitution **p**rinciple- מחלקות בן יכולות להיות מוחלפות במחלקת אב. הרעיון הוא שפונקציה שמקבלת אבא תוכל לקבל בן בלי בעיות מיוחדות. אם אנחנו למשל משתמשים בRTTI על אב שקיבלנו ובהתאם למי שהוא באמת נפעיל מתודה- הפונקציה הזו תצטרך התאמה לכל בן במיוחד וזה פוגע גם בעקרון OCP. כדי לבדוק אם אנחנו עומדים בLSP נכין פונקציה שמקבלת אב ונראה מה קורה כששולחים לה בן. עקרון זה מדבר יותר על יחסי ירושה של הרחבה ולא על אינטרפייס.

שאלת ראיונות עבודה: מלבן וריבוע- מי יורש ממי? מצד אחד ריבוע הוא סוג של מלבן, ומצד שני יש בריבוע פחות פרמטרים ולכן זה בזבוז להחזיק עבורו 2 פרמטרים.

ננסה לבדוק האם ירושת ריבוע ממלבן עומדת בעקרונות LSP ונראה שאם תהיה פונקציה שמציבה אורך ורוחב ומחשבת שטח- השטח יהיה (הגובה \* הגובה) עבור ריבוע וזו התנהגות לא צפויה. לכן ריבוע ומלבן ירשו שניהם מאב משותף ויהיו אחים.

1. ISP, **i**nterface **s**egragetion **p**rinciple- אם ירשנו מחלקה, זה אומר שאנחנו צריכים את כל מה שירשנו. זה לא נכון לרשת מישהו שאנחנו לא צריכים את כל האינטרפייס שלו. זה יוצר מצב ששינוי בפונקציונאליות שלא רלוונטית לגבי דורש ממני קימפול מחדש. דוגמא מהמצגת: יש מערכת ומשתמשים שונים משתמשים בה, כל אחד בפונקציונאליות אחרת. איך נדאג שכל אחד לא יירש פונקציונאליות שלא רלוונטית לגביו? נייצר לכל משתמש אינטרפייס משלו עם הפונקציונאליות שלו. כך מתאפשרת הסתרת public. המערכת הכללית תממש את כל האינטרפייסים ואז נוכל להתייחס למערכת כבן שנשלח בשם האינטרפייס המתאים כשנוכל להשתמש רק בפונקציונאליות הנצרכת. יש כאן ירושה ע"י ייצור אבא כדי להסתיר מידע.
2. **D**epenting **i**nvertion **p**rinciple,DIP- רוצים להפריד בין רמה עליונה ורמות אחרות. קלט פלט הוא low level. דברים טכניים. יש להפריד את זה מרמה לוגית. לכן מחלקה לוגית לא מתעסקת בקלט פלט אלא עוסקת בget set בלי שישנה לה כיצד מתבצעת התקשורת. עשויה להיווצר בעיה במידה ויש משהו לא בסדר המתגלה ברמת הלוגיקה כשאין ברמה זו תקשורת למשתמש. זריקת exeption תהיה בעייתית במקרה כזה כי היא כבדה. לכן נשתמש בinterface. נכין interface לקלט פלט ואם תהיה בעיה למחלקת הלוגיקה, היא תוכל לתקשר דרך האינטרפייס בלי תלות בדרך המימוש של הקלט פלט אלא דרך פוינטר לאינטרפייס.

**(תזכורת**: כשרוצים להוסיף פונקציונאליות למחלקה בלי לשנותה משתמשים בירושת private. מה שהיה לאב בpublic נכנס לprivate של הבן).

**מצגת 9- סביבות פיתוח**

עולם הפיתוח מתחלק ל2: user mode ו- kernel mode.

User mode- אלו התוכנות שאנו כותבים. תוכנות שנכתבות רמה אחת מעל שיחה ישירה עם חומרת המחשב. אנחנו מדברים עם כתובות לוגיות ולא פיזיות. בתכנות זה אתה לא יכול להרוס קוד של מישהו אחר כי יש מערכת הפעלה ששולטת עלינו ושומרת מפגיעה בחומרה. בשיטה זו יש לנו גישה מוגבלת למשאבים. הריצה דרך פונקציות system API של מערכת ההפעלה המקשרות לחומרה.

Kernel mode- kernel הוא ליבה. הרכיב המרכזי של מרבית מערכות ההפעלה. מערכת ההפעלה אחראית על חיבור בין תוכנה לחומרה. (ניתן לרוץ בלי מערכת הפעלה כמו למשל בקוצב לב בו הקוד נכתב ישירות לחומרה בלי חציצה של מערכת הפעלה בין הקוד לחומרה). זוהי ריצה embedded- ישירות על חומרה.

מבחינה היסטורית על מערכות הפעלה, בעבר לא היו רצים עם מערכת הפעלה וכל פקודה הוצרכה לכתוב עד הרמה הכי נמוכה. אח"כ פיתחו את unix שיכלה לרוץ רק על מחשבים גדולים (ונכתבה בשפת c), ואח"כ פותח לינוקס שמתאים למחשב PC פשוט ונכתב ע"י סטודנט כפרויקט סיום. אח"כ פותחו אבונטו ומערכות הפעלה שהוסיפו הצגה גרפית.

בkenel mode מדברים ישירות לזיכרון ולחומרה (מעבד, זיכרון, וקלט פלט בד"כ). לכן נדרשת הבנה טובה של החומרה. כשkenel לא רץ כמו שצריך- כולם נתקעים ולכן כל באג קריטי הרבה יותר מאשר user mode. כמו"כ יש פקודות ישירות של CPU שאפשר להריץ ועוסקים בכתובות פיזיות.

הdriver הוא האינטר פייס איך לדבר עם החומרה. מתרגם. הקוד של לינוקד הוא open sores ולכן כל אחד יכול לקחת אותו ולבצע עליו שינויים בkernel בהתאם לצרכיו.

עד כאן למבחן. מכאן, לידע כללי.

**סוגי תכנות:**

1. Short lived utilities- תוכנות קטנות שצריך לרגע ומבחינתנו לא נורא אם יהיו בהן באגים ונריץ אותן שוב.
2. Desktop application- תוכנות שאנחנו מצפים מהן להחזיק מעמד כמה ימים. יש יחסית סבלנות לקריסה אחת לכמה ימים (למשל word שבו מקסימום נסגור ונדליק אם יש בעיה פעם בכמה ימים).
3. Server,web,embedded,RT- אלו תוכנות שאי אפשר שהן לא יעבדו. כל הזמן הן צריכות לעבוד 24/7. כל הזמן צריכה להיות זמינות של תוכנות אלו. אנחנו לא נסבול אינטרנט שלא עובד שעתיים בחודש!

בהתאם לדרישות הנ"ל, יעשו הבדיקות לתוכנה וככל שהדרישה ממנה גבוהה יותר הבדיקות יהיו קפדניות יותר, ויידרש design קפדני ומדויק יותר.