תרגילי תכנות מתקדמים בסביבת Win32

סיון טולדו

31 באוגוסט 2004

האתר הזה מכיל תרגילים בתכנות מערכות (system programming) בסביבת חלונות. התרגילים מיועדים להיות מוטלים בקורס "מערכות הפעלה" באוניברסיטאות ומכללות, אבל הן גם מתאימים לקורסים דומים, כגון קורסים בתכנות מערכות, וללימוד עצמי.

בקורס אוניברסיטאי, התרגילים מיועדים להיות מוטלים בקצב של אחד בשבוע. היקף התרגילים ורמת הפירוט בתרגיל מאפשרות לתלמידים ותלמידות עם יכולת תכנות בשפת C לפתור אותם בקצב כזה. הכוונה היא שפתרון תרגיל יקח מספר שעות לכל היותר, אבל כמובן שרמת המאמץ של כל תלמיד ותלמידה תלוייה בנסיון התכנות שלהם. כמובן שאי אפשר להטיל את כל התרגילים בסימסטר אוניברסיטאי של 13 או 14 שבועות. באוניברסיטת תל־אביב, אנו מטילים בין 10 ל־12 תרגילים בסמסטר.

התרגילים תוכננו על פי מספר עקרונות:

- בוגרי מדעי המחשב (ומתכנתים מקצועיים בכלל) צריכים להכיר את כל מגוון השירותים שמערכת ההפעלה מספקת לתוכניות. היכרות כזו גם משרתת גם את הבנת מבנה מערכת ההפעלה וגם משפרת את מיומנויות התכנות של הבוגרים. אי לכך, התרגילים מכסים את רוב השירותים של מערכת ההפעלה, גם אם אינם מכסים כל קריאת מערכת (בחלונות יש מאות קריאות מערכת ואין זה מעשי לכסות את כולן).

- הדרך הטובה ביותר לכסות כמות חומר גדולה כזו היא על ידי רצף של תרגילים קטנים יחסית ולא על ידי פרוייקט גדול. התרגילים הקטנים מאפשרים להתמקד בנושא אחד בכל שבוע, והם גם מפזרים את העומס על התלמידים לאורך הסימסטר.

- על מנת לאפשר לתלמידים לפתור כל תרגיל תוך מספר שעות, התרגילים מספקים חלק גדול מהתיעוד לתוכניתן שנדרש באותו תרגיל. כלומר, קריאות המערכת ואספקטים אחרים של מנשק מערכת ההפעלה מתועדים בתוך התרגיל. התיעוד בגוף התרגיל מאפשר לתלמידים ללמוד את קריאות המערכת שרלוונטיות לאותו תרגיל באופן אינטנסיבי.

- יחד עם זאת, מיומנות העיון בתיעוד המקוון חשובה גם היא לתוכניתנים מקצועיים. על מנת לפתח מיומנות זאת, התרגילים משמיטים לעיתים חלק מהתיעוד הדרוש ומעודדים את התלמיד/ה לעיין בתיעוד המקוון. בתכנות מערכות בחלונות, התיעוד הוא חלק מ־Microsoft Developer Network (או MSDN) שזמין באינטרנט באתר msdn.microsoft.com. בדרך כלל, התיעוד גם מותקן בתחנות העבודה כחלק מכלי הפיתוח.

במסגרת פיתוח התרגילים, מימשנו כמובן פתרונות לכל התרגילים. את הפתרונות נשמח לחלק למרצים שמטילים את התרגילים במסגרת קורסים. הפתרונות אינם זמינים לתלמידים, ואנו מבקשים ממרצים אחרים לא להעמיד את הפתרונות שלנו לרשות תלמידים.

חומרי הלימוד הללו זמינים גם באנגלית.

התרגילים הוכנו על ידי סיון טולדו מבית הספר למדעי המחשב באוניברסיטת תל־אביב, שם הם הוטלו במסגרת קורסים במערכות הפעלה. התרגילים הוכנו בסיוע מענק לפיתוח תוכניות לימודים מחטיבת המחקר של חברת מייקרוסופט.

קריאה וכתיבה מקבצים בחלונות

מטרת תרגיל זה לתרגל כתיבת והרצת תוכנית פשוטה בחלונות ולתרגל שימוש בקריאות מערכת בסיסיות לגישה לקבצים.

מבנה תכנית C רגילה בחלונות (שלוש נקודות מציינות קטעים חסרים):

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <tchar.h>
...
int _tmain(int argc, LPTSTR argv[])
{
  if (argc < 2) { /* assuming the program needs 1 ar-
gument */
    _tprintf( _T("usage: %s <arg>\n"),
              argv[0]);

exit(1); /* stop the program; report error */  }
  ...
  return 0; /* successful return */
}
```

אפילו תוכנית פשוטה כל כך מפגינה שני צדדים מורכבים מעט של פיתוח בסביבת חלונות. המורכבות הראשונה נובעת מכך שבחלונות ניתן ליצור תוכניות שבהן תוים יכולים להיות בני 8 או 16 סיביות. כאשר תוים מיוצגים רק ב-8 סיביות, המשמעות של תו תלויה בקידוד של הקובץ: למשל, בקובץ טקסט עברי תו מספר 224 מייצג את האות אלף, אבל בקובץ טקסט צרפתי אותו תו מייצג אות לטינית. כאשר תוים מיוצגים ב-16 סיביות, מערכת ההפעלה משתמשת תמיד בקידוד בסטנדט בשם יוניקוד (Unicode) שבו יש יצוג נפרד לכל אות בכל שפה שהיא. בקידוד יוניקוד מחרוזת אחת יכולה להכיל טקסט בעברית, צרפתית, יפנית וערבית, למשל. על מנת שניתן יהיה לייצר מאותו קובץ מקור גם תוכניות שבהן תוים מיוצגים ב-8 סיביות וגם תוכניות שבהן תוים מיוצגים ב-16 סיביות, מייקרוסופט הגדירה טיפוס נתונים של תוים מוכללים, TCHAR. כאשר בונים את התוכנית ושני משתני ה־preprocessor שמוגדרים כאן בשורות הראשונות, UNICODE ו-_UNICODE, מוגדרים, משתנים מטיפוס TCHAR יתפסו שני בתים (16 סיביות), אבל כאשר המשתנים הללו לא מוגדרים, משתנים מטיפוס TCHAR יתפסו רק בית אחד. גם המשמעות של המקרו _T ושל פונקציות כגון _tprintf תשתנה בהתאם. כאן אנו מעוניינים בקידוד יוניקוד, אבל אם נסיר מהתוכנית את שתי השורות הראשונות נקבל תוכנית תקינה שבה תוים תופסים בית אחד בלבד.

על מנת לכתוב תוכניות תומכות יוניקוד (וכן תומכות בתוים בני בית אחד), יש להקפיד על הנקודות הבאות:

- הפונקציה הראשית בתוכנית היא _tmain ולא main.
- מחרוזות מפורשות יש לעטוף במקרו _T.
- תוים ומערכים של תוים יש להגדיר בעזרת הטיפוס TCHAR במקום char.

- יש לקרוא לפונקציות ספריה שמטפלות במחרוזות מתוך הספריה של מייקרוסופט ולא מתוך הספריה הסטנדרטית של שפת C.

על מנת להשלים את הדיון, כדאי לציין שגם רוב מערכות יוניקס ולינוקס תומכות ביוניקוד, אבל בגישה שונה. הגישה ביוניקס ולינוקס מבוססת בדרך כלל על קידוד של תווי יוניקוד במספר משתנה של בתים, בפורמט הנקרא UTF-8. היתרון בגישה זו הוא שמחרוזות שמועברות אל ומאת מערכת ההפעלה ממשיכות להיות מחרוזות של בתים בודדים (מערכים של char), וכן שהקידוד של מחרוזות ASCII אינו משתנה (כלומר של מחרוזות של ספרות, סימני פיסוק, ותווים לטיניים ללא סימני עזר). החסרון העיקרי של גישה זו לעומת הגישה בחלונות הוא שמספר הבתים במחרוזת אינו מלמד מה מספר האותיות במחרוזת ולהיפך.

התוכנית הפשוטה הזו מראה על עוד צד מורכב בתכנות בחלונות: טיפוסי משתנים רבים שהוגדרו מראש, ויש צורך להבין את משמעותם. הטיפוסים החשובים ביותר הם:

- משתנים בולאניים מטיפוס BOOL והקבועים המתאימים TRUE ו-FALSE.
- משתנים שלמים אי־שליליים בני 32 סיביות מטיפוס DWORD.
- מצביעים, שתבנית שמם היא *LP, כגון מצביע חסר טיפוס LPVOID, מצביע ל-TCHAR ששמו משום מה הוא LPTSTR, וכדומה.
- משתנים שמייצגים משאב כלשהו שמערכת ההפעלה העניקה גישה אליו, מסוג HANDLE. משתנים כאלה מייצגים קובץ פתוח, למשל.

כמעט כל תוכנית שדורשת גישה לקריאות המערכת של Win32 צריכה לכלול את הקבצים windows.h ו-tchar.h. מעתה ואילך לא נזכיר אותם, אבל הם תמיד דרושים.

חלק גדול מקריאות המערכת בחלונות מחזיר ערך בוליאני. הערך TRUE מציין הצלחה והערך FALSE מציין כישלון. ניתן לברר את סיבת הכישלון בעזרת קריאת המערכת GetLastError(), שמחזירה קוד שגיאה מטיפוס DWORD. שגרת הספריה FormatMessage מחזירה מחרוזת שמתארת את משמעות קוד השגיאה.

פתיחת קובץ: קריאת המערכת CreateFile פותחת קובץ קיים ו/או יוצרת קובץ חדש. היא מחזירה מזהה שבעזרתו ניתן לקרוא ולכתוב מהקובץ (ולבצע מספר פעולות נוספות), או את הערך INVALID_HANDLE_VALUE במקרה של כשלון.

```
HANDLE CreateFile(
       LPCTSTR filename,
       DWORD   access_flags,
       DWORD   share_mode_flags,
       LPSECURITY_ATTRIBUTES sa,
       DWORD   create_flags,
       DWORD   attributes_and_flags,
       HANDLE  template_file);
```

הארגומנט הראשון הוא שם הקובץ שרוצים לפתוח, למשל ..\data או C:\Temp\xyz.dat.

הארגומנט השני מתאר האם רוצים לפתוח את הקובץ לקריאה, כתיבה, או שתיהן. כל אחת מהפעולות מצויינת על ידי קבוע סימבולי,GENERIC_READ ו-GENERIC_WRITE. ניתן לבקש לבצע את שתיהן על ידי פעולת or של שני הקבועים.

הארגומנט השלישי מתיר או אוסר על פתיחת הקובץ לקריאה או כתיבה בזמן שהקובץ פתוח. הערך 0 אוסר על מערכת ההפעלה לפתוח את הקובץ (מתכניות אחרות או אפילו

פעם נוספת מהתוכנית הזו) עד שאנו נסגור אותו, והערכים FILE_SHARE_READ ו-FILE_SHARE_WRITE מתירים לתוכניות אחרות לפתוח אותו לקריאה ו/או כתיבה בזמן שהוא פתוח על ידינו.

הארגומנט הרביעי קשור בהרשאות ובינתיים נשתמש בערך NULL, שמציין שיש להשתמש בברירות המחדל.

הארגומנט החמישי מתאר מה לעשות אם הקובץ קיים, או אם הוא איננו:

- הערך CREATE_NEW גורם ליצירת קובץ חדש אם אין קובץ בשם זה, ולכשלון אם הקובץ קיים.
- הערך CREATE_ALWAYS גורם ליצירת קובץ חדש, בין אם היה קובץ בשם זה או לא.
- הערך OPEN_EXISTING גורם לכשלון אם הקובץ לא קיים.
- הערך OPEN_ALWAYS גורם לפתיחת קובץ קיים או ליצירת קובץ חדש.
- הערך TRUNCATE_EXISTING גורם לקיצוץ קובץ קיים לגודל אפס, ולכשלון אם אין כזה קובץ.

הארגומנט השישי מציין תכונות של קובץ חדש (במקרה שאכן נוצר קובץ חדש) ומאפשר לבקש או להמליץ על מנגנוני גישה מיוחדים. בינתיים נשתמש בערך ברירת המחדל שהוא FILE_ATTRIBUTE_NORMAL. בין היתר, ניתן לבקש בעזרת ארגומנט זה שהקובץ החדש ימחק כאשר נסגור אותו, שקריאות מערכת שכותבות לקובץ לא יחזרו עד שהמידע לא יכתב פיזית לדיסק, ולהכריז שגישות לקובץ הן סדרתיות בעיקר או שאינן סדרתיות.

הארגונמט השביעי מאפשר ליצור קובץ חדש עם מאפיינים של קובץ קיים פתוח. אנו לא נשמש בינתיים ביכולת זו ונעביר את הערך NULL.

סגירת קובץ או החזרת משאב אחר המיוצג על ידי handle:

```
BOOL CloseHandle(HANDLE h);
```

הפרמטר הוא המזהה שהוחזר על ידי CreateFile, או על ידי פונקציה אחרת שהחזירה מזהה מטיפוס זה. הפונקציה מחזירה קוד שגיאה. בדרך כלל סיבת הכישלון האפשרית היחידה היא שהועבר מזהה לא חוקי, כלומר מזהה שלא מייצג קובץ פתוח או משאב אחר שהוקצה על ידי מערכת ההפעלה.

קריאה וכתיבה:

```
BOOL ReadFile (HANDLE  h,
                  LPVOID  buffer,
                  DWORD   number_of_bytes_to_read,

LPDWORD number_of_bytes_actually_read,
                  LPOVERLAPPED overlapped);
BOOL WriteFile(HANDLE  h,
                  LPVOID  buffer,
                  DWORD   number_of_bytes_to_read,

LPDWORD number_of_bytes_actually_read,
                  LPOVERLAPPED overlapped);
```

האגומנט הראשון הוא מזהה של קובץ פתוח והמזהה השני הוא מצביע למערך שאותו יש להעביר אל או מתוך הקובץ. הארגומנט השלישי הוא מספר הבתים שהתוכנית מבקשת להעביר, והרביעי הוא מצביע למשתנה שיכיל, לאחר חזרת קריאת המערכת, את מספר הבתים שהועברו בפועל. המספר הזה עשוי להיות קטן יותר מהמספר שביקשנו להעביר אם הגענו לסוף הקובץ בבקשת קריאה, או שאין מקום בדיסק בבקשת כתיבה. הארגומנט האחרון משמש לצורת גישה לקבצים שלא נדון בה כעת, ונעביר בינתיים את הערך NULL.

לכל קובץ פתוח יש מצביע שמגדיר את המקום בקובץ (בבתים) שבו תתבצע הקריאה או הכתיבה הבאה. כל פעולת קריאה וכתיבה מקדמת את המצביע במספר הבתים שהועברו. כאשר הקובץ נפתח המצביע מצביע לתחילת הקובץ.

הערות לגבי fopen, fread, וכדומה. הפונקציות `fopen`, `fread`, `fwrite`, `fseek` מספקות שירותים דומים לקריאות המערכת שמתוארות כאן אך הן חלק מהספריה הסטנדרטית של C ולא קריאות ישירות למערכת ההפעלה. שימוש בהן הופך תוכנית ליותר פורטבילית אך לא ניתן לבצע דרכן כל פעולה שניתן לבצע דרך הממשק הישיר למערכת ההפעלה (בגלל שוני בין מערכות הפעלה). בספר זה נשתמש בממשק הישיר למערכת ההפעלה ולא בספריות עזר.

התרגיל. כתוב/כיתבי תכנית שמעתיקה קובץ. התכנית צריכה לצאת ולהדפיס הודעת שגיאה אם אין מספיק פרמטרים לתכנית, אם קובץ הקלט אינו קיים, ואם קובץ הפלט קיים. התכנית צריכה להתשמש בקריאות המערכת המתוארות בתרגיל. ההעתקה צריכה להתבצע תוך שימוש בחוצץ שגודלו n בתים, כלומר התכנית קוראת n בתים בקריאה אחת, כותבת אותם וחוזר חלילה. לתכנית שלושה פרמטרים, שם קובץ הקלט, הפלט, ו-n. על מנת להפוך את המחרוזת המכילה את הייצוג של n לשלם אפשר להשתמש ב-`_stscanf(argv[3],_T("%d"),&n)`. יש להשתמש ב-`malloc` על מנת להקצות את הזיכרון לחוצץ:

```
#include <stdlib.h>
...
char* buffer;
...
buffer = (char*) malloc(n);
```

מידדו את זמן הריצה של התוכנית בעזרת הפקודה `processtimes` (באתר) כאשר אתם משתמשים בתוכנית להעתיק קובץ בגודל מליון בתים או יותר. מידדו את זמני הריצה תוך שימוש בחוצץ בגדלים הבאים (בבתים): 1, 64, 512, 1024, 8192, ו-65536. אם יש שוני ניכר בין זמני הריצה עם חוצץ בגדלים שונים, יש להסביר את הסיבות לשוני.

יצירת תהליכים והרצת תוכניות

בתרגיל זה נלמד ליצור תהליך חדש ולהריץ בו תוכנית. דרך הפעולה של התוכנית שנכתוב דומה מאוד לדרך הפעולה של תוכניות מעטפת (shell) כגון cmd.exe, אם כי תוך שימוש בממשק פשוט בהרבה שאינו דורש פענוח של שורת פקודה מורכבת.

יצירת תהליך חדש:

```
BOOL CreateProcess(LPCTSTR path,
                   LPTSTR  command_line,
                   LPSECURITY_ATTRIBUTES process_sa,
                   LPSECURITY_ATTRIBUTES thread_sa,
                   BOOL    inherit_handles,
                   DWORD   creation_flags,
                   LPVOID  environment,
                   LPTSTR  current_directory,

LPSTARTUPINFO         startup_info,

LPPROCESS_INFORMATION process_info);
```

קריאת המערכת יוצרת תהליך חדש ומריצה בו תוכנית. למעשה, הקריאה יוצרת תהליך וחוט–התהליך הוא מעין מחשב וירטואלי עם זיכרון פרטי, והחוט הוא מעבד וירטואלי במחשב הוירטואלי. שני הארגומנטים הראשונים מתארים את התוכנית שאנו מבקשים להריץ ואת הפרמטרים (שורת הפקודה) שלה. אם הארגומנט הראשון אינו NULL, אז הוא חייב להכיל שם של קובץ שמבקשים להריץ. במקרה כזה, מערכת ההפעלה אינה מחפשת תוכנית מתאימה, היא פשוט מפעילה את קובץ הריצה ששמו נתון. הארגומנט השני מכיל את שורת הפקודה שהתוכנית תקבל. אם הארגומנט הראשון הוא NULL, אז מערכת ההפעלה מתייחסת לארגומנט השני כאל שורה שמוקלדת בתוכנת המעטפת (shell), בדרך כלל cmd.exe. במקרה כזה, מערכת ההפעלה מחפשת תוכנית ששמה הוא המילה הראשונה בשורת הפקודה; החיפוש מתבצע במדריך שממנו הופעלה התוכנית שביצעה את קריאת המערכת, במדריך הנוכחי של התוכנית, במדריכים של מערכת ההפעלה עצמה, ובמדריכים ששמם מופיע במשתנה הסביבה PATH.

שני הארגומנטים הבאים מאפשרים לקבוע הרשאות גישה לתהליך ולחוט שייווצרו. הערך NULL מציין בקשה לשימוש בברירות מחדל. הארגומנט הבא מציין האם אנו מעוניינים שהתהליך החדש יירש את המזהים של קבצים פתוחים ומשאבים אחרים ויוכל להשתמש בהם.

הארגומנט השישי מאפשר לשלוט בצורה שבה ייווצר התהליך החדש. ניתן ליצור את התהליך כך שירוץ בחלון הקיים (של ההורה שיוצר אותו), בחלון חדש, או ללא חלון כלל. ניתן ליצור את התהליך החדש כך שיהיה חלק מקבוצת התהליכים שגם ההורה שייך אליה, או שיתחיל קבוצה חדשה. קבוצת התהליכים משפיעה על תגובה לאירועים חיצוניים כגון נסיון להפסיק ריצה על ידי הקשת control-c. יש פרמטרים נוספים שניתן לשלוט בהם דרך הארגומנט הזה. לפרטים נוספים, ראו בתיעוד המקוון. הערך 0 משתמש בברירות מחדל נוחות למקרים פשוטים.

הארגומנט השביעי מצביע לשטח זיכרון שמתאר את משתני הסביבה שהתוכנית תוכל לגשת אליהם. משתני הסביבה הם המשתנים שניתן לקבוע את ערכם ולשלוף את ערכם בעזרת הפקודה set במעטפת, וניתן לקרוא אותם גם מתוך תוכניות. משתני הסביבה מתוארים על ידי סדרת מחרוזות מהצורה name=value שכל אחת מהן מסתיימת ב-0, הן משורשרות ברצף

אחת אחרי השניה, ואחרי האחרונה יש ערך 0 נוסף. המחרוזות הללו הן בדרך כלל של תווי ASCII, אלא אם דגל בארגומנט הקודם ציין שהמחרוזות הן מחרוזות יוניקוד. הערך NULL גורם לירושת משתני הסביבה של ההורה. הארגומנט השמיני מאפשר להתחיל את התהליך החדש במדריך אחר מזה שהתהליך היוצר משתמש בו. גם כאן הערך NULL גורם לירושה מהתהליך היוצר.

הארגומנט התשיעי מאפשר לשלוט על שני דברים. הראשון, המראה של החלון שבו ירוץ התהליך החדש, אם הוא ירוץ בחלון חדש. השני, על ערוצי הקלט, פלט, ושגיאה הסטנדרטיים של התהליך החדש (stdin, stdout, ו-stderr בספריה הסטנדרטית של שפת C). אם לא קובעים ערוצים כאלה, התהליך החדש ישתמש בערוצים שמחוברים לחלון שבו הוא רץ, אם הוא אכן רץ בחלון טקסט. **הערך** NULL **אינו חוקי כאן**; יש להעביר כתובת של מבנה מטיפוס STARTUPINFO, שהשדה cb שלו מכיל את גודלו בבתים. שאר השטח יכול להכיל אפסים אם מעוניינים להשתמש בברירת המחדל. אפשר להשתמש בשגרה ZeroMemory לצורך איפוס השטח, אבל ניתן גם להשתמש בלולאה פשוטה.

הארגומנט האחרון מחזיר מידע על התהליך והחוט החדש שנוצרו במבנה מטיפוס PROCESS_INFORMATION. משום מה, יש לאפס גם את המבנה הזה לפני הקריאה. המבנה מכיל ארבעה שדות: מזהים פתוחים (מטיפוס HANDLE) לתהליך ולחוט, ששמותיהם hProcess ו-hThread, והמזהה המספרי שלהם, dwProcessId ו-dwThreadId. המזהים המספריים הם למעשה השמות של התהליך והחוט, ובעזרתם ניתן לבצע פעולות כגון הפסקת פעולה של תהליך. ניתן לראות את המזהה של כל תהליך (Process Identifier או PID) ב-task manager; השדה הזה אינו מוצג בדרך כלל, אך ניתן לבחור אותו להצגה. את המזהים הפתוחים יש לסגור באמצעות CloseHandle כאשר לא זקוקים להם יותר.

המתנה (כמעט לכל דבר). במערכות חלונות יש קריאות מערכת שגורמות לתוכנית לחכות לאירוע כלשהוא, כגון סיום פעולת חוט, סיום פעולת תהליך, ועוד. בחלונות, כל עצם יכול להמצא באחד משני מצבים: מסומן (signaled) או לא מסומן. חוט או תהליך אינם מסומנים כל זמן שהם רצים או עשויים לרוץ בעתיד, ומסומנים כאשר הם מסיימים את פעולתם.

קריאת המערכת שנשתמש בה על מנת להמתין שעצם יגיע למצב מסומן היא WaitForSingleObject. הקריאה ממתינה שחוט או תהליך יסיים את פעולתו, או שעצם מסוג אחר יגיע למצב מסומן.

```
DWORD WaitForSingleObject(
      HANDLE object,
      DWORD  timeout_in_milliseconds);
```

הארגומנט הראשון הוא המזהה של העצם שמבקשים לחכות לאירוע הקשור בו, והארגומנט השני הוא זמן ההמתנה המקסימלי באלפיות שניה, או הקבוע INFINITE אם ההמתנה אינה מוגבלת בזמן. השגרה מחזירה אחד משלושה ערכים: WAIT_OBJECT_0 אם ההמתנה הסתיימה בהצלחה, WAIT_TIMEOUT אם ההמתנה הסתיימה בגלל שפרק הזמן המקסימלי שציינו עבר, או WAIT_ABANDONED, שיכול להיות מוחזר רק כאשר העצם הוא מנעול (mutex).

זמן הריצה של תהליך. הקריאה GetProcessTimes מחזירה מידע לגבי תקופת הריצה של תהליך ומשאבי המעבד שהוא צרך.

```
BOOL GetProcessTimes(HANDLE     process
                     LPFILETIME creation_time,
                     LPFILETIME exit_time,
                     LPFILETIME kernel_time,
                     LPFILETIME user_time);
```

הארגומנט הראשון הוא המזהה של התהליך שמבקשים מידע אודותיו. כל שאר הארגומנטים הם מבנים מסוג FILETIME שמתארים תקופת זמן ביחידות של 100 ננו־שניות, בעזרת 64 סיביות. הסיביות הללו מחולקות לשתי מילים של 32 סיביות, dwLowDateTime ו-dwHighDateTime. הארגומנטים השני והשלישי מתארים נקודת זמן ספציפית, של יצירת התהליך וסיום הריצה שלו; תקופת הזמן שהמבנה מחזיק היא התקופה שמנקודת ציון קבועה בעבר (תחילת שנת 1601 באיזור הזמן של גריניץ', אנגליה) ועד לנקודת הזמן המתוארת. השגרה FileTimeToSystemTime ממירה ייצוג כזה לייצוג שמכיל תאריך ושעה באופן מפורש. שני הארגומנטים האחרונים מתארים פשוט כמות זמן, את הכמות שהתהליך צרך על המעבד במצב מיוחס ובמצב משתמש.

הערך המוחזר מתהליך. תהליך יכול להחזיר ערך שלם כאשר הוא מסיים את פעולתו. הערך הזה משמש בדרך כלל לציון הצלחה או כישלון במשימה שהתהליך ניסה לבצע. הערך יכול להיות מוחזר על ידי הפקודה return מהשגרה main בתוכנית, על ידי קריאה לפונקציית הספריה exit של שפת C, או על ידי קריאות המערכת ExitProcess ו-TerminateProcess.

```
BOOL GetExitCodeProcess(HANDLE  process
                        LPDWORD exit_code);
```

הארגומנט הראשון הוא המזהה של התהליך שמבקשים מידע אודותיו, והשני הוא מצביע למשתנה שיכיל את הערך שהתהליך החזיר אם הוא סיים את פעולתו או הערך הסימבולי STILL_ACTIVE אם הוא עדיין רץ או עשוי לרוץ.

במעטפת (cmd.exe), ניתן לגשת לערך המוחזר מהתוכנית האחרונה שהמעטפת הריצה או ניסתה להריץ דרך המשתנה %errorlevel%, למשל

```
c:\> lpq -Smambo -Phpintel
hpintel@mambo 1 job
c:\> echo %errorlevel%
0
c:\> nosuchprogram
'nosuchprogram' is not recognized as an inter-
nal or external command,
operable program or batch file.
c:\> echo %errorlevel%
9009
c:\>dir l:
The system cannot find the path specified.
c:\> echo %errorlevel%
1
```

התרגיל. עליך לכתוב תוכנית בשם ex-processtimes המפעילה תכנית אחרת ומדפיסה את הערך המוחזר ממנה ואת זמן הריצה שלה לאחר שהיא מסיימת. התכנית צריכה לקבל מהמשתמש שם תוכנית וארגומנטים על שורת הפקודה, להריץ את התוכנית, להמתין שתסיים, ולאחר מכן להדפיס את זמני הריצה של התוכנית שהורצה. הפעלה לדוגמה של התכנית:

```
c:\> ex-processtimes c:\windows\sysem32\lpq.exe -
Smambo -Phpintel
```

```
hpintel@mambo 0 jobs

error level  0
elapsed time 0.110 seconds
user time    0.010 seconds
kernel time  0.040 seconds
```

תרגיל תכנות: שימוש באיתותים (signals)

עדיין דורש עדכון לסביבת חלונות!!!

איתותים (signals) בלינוקס ויוניקס הינם פסיקות וירטואליות שמערכת ההפעלה שולחת לתהליך במצבים מסויימים, כמו למשל נסיון לגשת לזיכרון לא ממופה, הודעה שהמערכת עומדת להסגר, ועוד. תכנית שמעוניינת לטפל באחד המצבים האלה מודיעה למערכת ההפעלה איזו שגרה להפעיל כאשר קורה המצב הזה. למשל, תכנית יכולה לבקש ממערכת ההפעלה להפעיל שגרה מסויימת בתכנית כאשר התכנית מנסה לגשת לזיכרון לא ממופה, על מנת לנסות למפות קובץ לדף שגרם לחריג הדף. כאשר שגרה כזאת מסיימת את פעולתה, מערכת ההפעלה ממשיכה להריץ את התהליך מאותו מקום שבו קרה המצב המיוחד. (במקרה של גישה לזיכרון לא ממופה, למשל, מערכת ההפעלה תריץ את התכנית החל מאותה גישה לזיכרון שנכשלה–אם הזיכרון עדיין אינו ממופה הפעולה תכשל שוב).

לכל מצב מיוחד כזה מערכת ההפעלה מגדירה התנהגות נורמלית למקרה שתכנית לא מודיעה על שגרה לטיפול בבעיה. ההתנהגות הנורמלית היא התעלמות ממצבים לא קריטיים והפסקת פעולת התכנית במקרים קריטיים. יש מצבים בעייתיים במידה כזו שמערכת ההפעלה לא מרשה לתכנית לשנות את ההתנהגות הנורמלית–העפת התהליך.

רישום שגרת טיפול באיתות:

```
#include <signal.h>
...
void (*signal(int signum, void (*han-
dler)(int)))(int);
```

קריאת המערכת `signal` מודיעה למערכת ההפעלה על השגרה שיש להפעיל כאשר מתקבל איתות מסוים. לקריאה יש שני ארגומנטים והיא מחזירה מצביע. הארגומנט הראשון הוא האיתות. בדרך כלל אין לא מציינים בתכנית ערך שלם מספרי אלא קבוע סימבולי כגון SIGSEGV או SIGTERM. שמות כל האיתותים מופיעים ב-`signal` `7` `man`. הארגומנט השני הוא כתובת של שגרת הטיפול באיתות. המצביע שהקריאה מחזירה מצביע לשגרה שהייתה אחראית לטיפול באיתות עד כה (החזרת ערך זה מאפשרת למודול בתכנית לשנות את שגרת האיתות בזמן שהמודול פועל ולהחזיר את ההתנהגות הקודמת עם היציאה מהמודול, ללא שהמודול יצטרך לדעת בדיוק מה הייתה ההתנהגות הקודמת).

שגרות טיפול באיתות מקבלות ארגומנט אחד, ערך שלם, ואינן מחזירות כל ערך. הערך שהן מקבלות הוא מספר האיתות, ממשק שמאפשר לשגרה אחת לטפל במספר איתותים.

במקום כתובת של שגרה ניתן גם להעביר את הערכים SIG_IGN ו-SIG_DFL. הערך הראשון מודיע למערכת ההפעלה שהתכנית רוצה להתעלם מהאיתות, והשני מודיע שההתנהגות הרצויה היא התנהגות ברירת המחדל.

סוגי איתותים. סוגי האיתותים ושמותיהם מופיעים ב-`signal` `7` `man`. בדף זה גם מצויינת ההתנהגות הרגילה של כל איתות.

לצורך תרגיל זה חשוב לדעת על SIGSEGV, איתות שנשלח כאשר התכנית מנסה לגשת לכתובת לא ממופה, על SIGTERM שנשלח כאשר מערכת ההפעלה מעוניינת שהתהליך יסיים את פעולתו, ועל SIGKILL שנשלח כאשר התהליך חייב להסתיים מייד.

שליחת איתות:

```
#include <signal.h>
#include <sys/types.h>
...
int kill(pid_t pid, int sig);
```

קריאת המערכת kill שולחת את האיתות sig לתהליך שמספרו pid. בדרך כלל נציין את האיתות בעזרת קבוע סימבולי ולא על ידי מספר.

הפקודה kill מאפשרת לשלוח איתותים לתהליכים באופן אינטראקטיבי. הפקודה מקבלת פרמטר מספרי אופציונלי שלפניו מופיע הסימן - המציין את מספר האיתות ואחריו מספרי תהליכים שיש לשלוח להם את האיתות. הפקודה kill 9- 1234, למשל, שולחת איתות מספר 9 (SIGKILL) לתהליך מספר 1234. כאשר לא מציינים את מספר האיתות, ברירת המחדל היא SIGTERM, בקשה לסיום פעולת התהליך.

שימוש ב-gdb. במסגרת התרגיל נכתוב תכנית המפעילה מנפה (debugger) בשם gdb. על מנת של-gdb יהיה מספיק מידע לניפוי התכנית (שמות משתנים וכתובותיהם, מספרי שורות, וכו'), יש לקמפל את התכנית עם הדגל g-. ניתן להפעיל תכנית תחת gdb על ידי מתן הפקודה gdb PROG (כאשר שם התכנית הוא PROG). ניתן גם להתחיל לנפות תכנית רצה על ידי מתן הפקודה gdb PROG PID כאשר PID הוא מספר התהליך של התכנית הרצה. הממשק של gdb הוא ממשק אלפא-נומרי שכולל מספר פקודות, שהשימושיות ביותר מתוכן הן:

פקודה	שימוש
	תחילת הרצת התכנית
	המשך ריצה מעצירה
	היכן אנו בתכנית
	הדפסת ערך משתנה
	הריגת התכנית שבודקים
	יציאה
	עזרה

התרגיל. זהו תרגיל מורכב יחסית שבו נלמד לטפל באיתותים. טיפול מעניין במיוחד באיתותים שאותו נתרגל הוא הכנסת התכנית לריצה תחת מנפה כאשר מתעורר מצב חריג בתכנית.

שימו לב לכך שגיאות תכנות בחלק האחרון של התרגיל (5-6) עשויות לגרום ליצירת מספר רב מאוד של תהליכים. בשל זאת, יש להשתדל יותר מתמיד להמנע משגיאות. במקרה של יצירת מספר רב של תהליכים, אנא הרוג אותם מחלון אחר. אם יש באפשרותך לפתח את התרגיל על מחשב שבשימושך בלבד, אנא עשה זאת. יש להזהר אך אין צורך לחשוש! דרך אחת להגביל את מספר התהליכים שעשויים להווצר בלינוקס הוא להקטין את החסם על מספר התהליכים שניתן ליצור בעזרת הפקודה limit maxproc X, כאשר X הוא מספר התהליכים שניתן ליצור בו זמנית תחת המעטפת (shell).

מפאת מורכבות התרגיל, יש לפתח את התכנית על פי השלבים הבאים (ולהגיש גם תשובות לשאלות המופיעות):

1. כתוב תכנית בשם ex-sig.c המדפיסה את המחרוזת ?continue ומחכה לקלט y. במקרה של כל קלט אחר יש להדפיס את השאלה שוב ולחכות לתשובה. לאחר קבלת הקלט הרצוי התכנית ממשיכה. הפעולה הבאה שלה היא השמת ערך כלשהו בכתובת o שלעולם אינה ממופה (למשל על ידי הפקודה *pointer=1 כאשר pointer==NULL.

2. הפעל את התכנית וכאשר היא מחכה לקלט שלח לה SIGTERM מחלון אחר בעזרת הפקודה kill. את מספר התהליך ניתן לגלות בעזרת הפקודה ps aux. האיתות הורג את התהליך.

3. כעת הוסף לתוכנית שגרת טיפול באיתותים. השגרה צריכה לבדוק את סוג האיתות, ובמקרה של SIGTERM פשוט להדפיס I will survive. בתכנית הראשית הוסף קוד שרושם את שגרת הטיפול הזו לאיתות SIGTERM. נסה כעת להרוג את התהליך על ידי הפקודה kill. מה קורה?

4. נסה להרוג את התהליך על ידי שימוש בפקודה kill -9 (SIGKILL). מה קורה כעת? האם ניתן לטפל במקרה זה כמו ב-SIGTERM?

5. כעת הוסף קוד שרושם את שגרת הטיפול גם עבור טיפול ב-SIGSEGV והוסף לשגרה בדיקה שתדפיס את אותה המחרוזת גם עבור איתות זה. מה קורה כאשר התוכנית מגיעה להשמה לכתובת 0?

6. כעת נשנה את התנהגות שגרת הטיפול כך שתפעיל את gdb כאשר התכנית מקבלת איתות על חריג דף (SIGSEGV). במקרה כזה השגרה צריכה לברר את מספר התהליך באמצעות קריאה ל-getpid (מוגדרת ב-unistd.h), וליצור תהליך חדש זהה באמצעות fork. התהליך החדש (הבן) צריך להפעיל בעזרת execv את הפקודה /usr/bin/gdb ex-sig PID כאשר במקום PID צריך להופיע מספר התהליך. התהליך המקורי צריך פשוט לחכות שהמנפה יתחבר אליו בעזרת sleep.

7. הפעל את התכנית וגרום לה לחריג דף. באיזה נקודה בתכנית מתחבר אליה המנפה? מה קורה כאשר מבקשים מהמנפה להמשיך את ריצת התכנית?

הערות למרצה: עבודות וחבורות תהליכים תחת חלונות

שני התרגילים הבאים ממששים את אותה פונקציונליות תוך שימוש בשני מנגנונים דומים אך נפרדים של Win32. בשני התרגילים המטרה היא להריץ תוכנית בתוך חבורת תהליכים חדשה, ולאחר מכן להרוג את כל התהליכים שבחבורה. תרגיל אחד משתמש בעבודות (jobs) והשני בחבורות תהליכים (process groups). המנגנון הראשון כללי יותר, ואילו השני דומה יותר לממשק של posix. בכל מקרה כדאי להטיל לכל היותר אחד משני התרגילים ולא את שניהם בקורס אחד.

התרגילים משתמשים בתוכנית עזר מסופקת שתפקידה לייצר תהליכים רבים עם אורך חיים קצר, על מנת שהתלמידים יוכלו לבדוק את הפתרונות שלהם. הרצת תוכנית העזר הזו עלולה לגרום לחוסר משאבים במערכת ההפעלה, שמתבטא באי יכולת ליצור תהליכים חדשים. כדאי ליידע את התלמידים על כך (אבל לדעתי כדאי להמליץ להם להתנסות בתופעה). לא כדי להריץ את תוכנית העזר הזו על שרת שבו חוסר המשאבים עלול לגרום להפסקת שירות למשתמשים אחרים פרט לתלמיד/ה עצמו/עצמה.

התרגילים משתמשים באירוע (event) על מנת לסמן שהגיע הזמן להרוג את התהליכים, ולכן כדאי להטיל את התרגילים הללו לאחר הטלת התרגילים בנושאי סינכרון וחוטים ולא לפניהם.

ניהול עבודות

מערכות הפעלה מאפשרות בדרך כלל לייצג באופן מפורש קבוצות של תהליכים ולבצע פעולות על כל התהליכים של קבוצה בבת אחת. קבוצה כזו, שנקראת חבורת תהליכים (process group) או עבודה (job) נוצרת באופן מפורש על ידי קריאת מערכת, וניתן לשייך אליה תהליך או תהליכים. כאשר תהליך שמשוייך לחבורה כזו יוצר תהליך נוסף, התהליך החדש משוייך אוטומטית לחבורה (אלא אם משייכים אותו מפורשות לחבורה אחרת). המנגנון הזה גורם לכך שניתן לשייך לחבורה עץ תהליכים שלם, שבו תהליכים יוצרים תהליכים אחרים, גם אם התהליכים הללו אינם משייכים את עצמם לחבורה באופן מפורש. תוכניות מעטפת (shells), למשל, משתמשות במנגנון הזה על מנת לאפשר למשתמש להפסיק את פעולתה של תוכנית שהריץ דרך המעטפת, גם אם התוכנית הזו יצרה מספר גדול של תהליכים, שהמעטפת אינה מודעת כלל לקיומם, ושהתהליך או התהליכים שיצרו אותם אולי כבר אינם קיימים.

בחלונות יש שני סוגים של חבורות תהליכים. סוג אחד נקרא חבורת תהליכים (process group). הסוג הזה דומה לחבורות תהליכים ביוניקס ולינוקס. יוצרים חבורה כזו על ידי ניתוק של תהליך מהחבורה שהוא שייך אליה; זה יוצר חבורה חדשה שהתהליך שנותק שייך אליה וששמה הוא מספר התהליך. הסוג הזה מוגבל למדי, מכיון שניתן להשתמש בו רק כאשר כל התהליכים בחבורה מחוברים לקונסולה אחת. מכיון שתוכניות גרפיות ותוכנות שרת בדרך כלל אינן מחוברות לקונסולה, לא תמיד אפשר להשתמש בסוג הזה.

הסוג השני נקרא עבודה (job), הוא יותר כללי, ובו נשתמש בתרגיל זה.

שיוך תהליך לעבודה:

```
HANDLE CreateJobObject       (LPSECU-
RITY_ATTRIBUTES sa,

LPCTSTR                name);
BOOL AssignProcessToJobObject(HANDLE job,
                              HANDLE process);
```

קריאת המערכת הראשונה יוצרת עבודה עם הרשאות נתונות (NULL עבור ברירת המחדל) ועם שם נתון. הקריאה מחזירה מזהה לעבודה. כאשר העבודה נוצרת, שום תהליך אינו משוייך אליה (גם לא התהליך שיצר אותה). קריאת המערכת השניה משייכת תהליך לעבודה, כאשר גם התהליך וגם העבודה מצויינים על ידי מזהה פתוח, לא על ידי שם העבודה ולא על ידי מספר התהליך.

כאמור, מטרת השיוך היא בדרך כלל לדאוג שהתהליך וגם כל צאצאיו יהיו משוייכים לעבודה. אולם אם יוצרים תהליך באופן רגיל ולאחר שהוא נוצר משייכים אותו, התהליך החדש עלול ליצור תהליכים נוספים לפני שנספיק לשייך אותו לעבודה. במקרה כזה, לאחר שיוכו הוא אמנם יהיה משוייך, אבל הצאצאים שכבר יצר לא ישוייכו לעבודה. על מנת למנוע מצב כזה, צריך ליצור את התהליך החדש כך שלא יוכל לרוץ, לשייך אותו לעבודה, ורק אז להתיר לו להתחיל לרוץ. כדי שהתהליך לא יתחיל לרוץ מייד עם יצירתו, יש ליצור אותו עם הדגל CREATE_SUSPENDED. כדי לאפשר לו להתחיל לרוץ לאחר שיוכו לעבודה, יש לקרוא לקריאת המערכת

```
DWORD ResumeThread(HANDLE thread);
```

עם ארגומנט שהוא מזהה של החוט הראשי של התהליך. במקרה של כשלון הקריאה הזו מחזירה 1-. (חוט יכול להיות מושעה מריצה בגלל מספר סיבות; הקריאה הזו מקטינה את מספר הסיבות באחד ומחזירה את מספר הסיבות שנותרו, 0 במקרה שהחוט יכול כעת להמשיך לרוץ.)

הפסקת ריצה של עבודה שלמה:

```
BOOL TerminateJobObject(HANDLE job,
                        UINT   exit_code);
```

הקריאה מפסיקה את פעולתם של כל התהליכים שמשוייכים לעבודה שהמזהה שלה נתון. המזהה הוא המזהה שהתקבל מהקריאה `CreateJobObject` או מהקריאה `OpenJobObject` שפותחת עבודה על פי שמה. הארגומנט השני הוא הערך שיוחזר מכל התהליכים שבעבודה.

התרגיל. עליך לכתוב תוכנית בשם `ex-job` שניתן להפעיל בשתי דרכים. כאשר מפעילים אותה עם שני ארגומנטים או יותר, היא יוצרת עבודה חדשה ויוצרת תהליך חדש ששורת הפקודה שלו היא הארגומנטים השני ואילך של `ex-job`. לפני שהיא יוצרת את התהליך החדש, היא יוצרת אירוע מסוג איפוס ידני ששמו הוא הארגומנט הראשון של `ex-job` ושמאותחל למצב לא מסומן. לאחר שהתהליך החדש התחיל לרוץ, התוכנית מחכה לאירוע, וכאשר האירוע מסומן, היא מפסיקה את פעולת כל התהליכים שמשוייכים לעבודה.

כאשר מפעילים את התוכנית עם ארגומנט אחד בלבד, היא פותחת את האירוע ששמו הוא הארגומנט הראשון שלה באמצעות קריאת המערכת `OpenEvent` ומסמנת אותו, מה שאמור לגרום לעבודה שהתחלנו אולי קודם לכן להפסיק לרוץ.

הנה הפעלה לדוגמה של התכנית:

```
                ↓ the name of the event
c:\> ex-job job-event netstat.exe -e 5
                              ↑ the command line to run in the new job
```

כדי להפסיק את פעולת העבודה, מפעילים מתוך חלון אחר את הפקודה

```
c:\> ex-job job-event
```

כדי לבדוק את התוכנית, השתמשו בתוכנית `CloneProcess` שמסופקת יחד עם התרגיל. צריך להריץ את התוכנית הזו עם ארגומנט אחד, משך זמן בשניות. התוכנית יוצרת שני תהליכים בהפרש של משך הזמן הנתון ואז מסיימת את פעולתה. לאורך זמן, האפקט הוא של יצירת מספר אקספוננציאלי של תהליכים, כאשר התהליכים שיצרו אותם כבר אינם קיימים. צפו ביצירת התהליכים הללו בעזרת מנהל המשימות ונסו לסיים את פעולתם בעזרת מנהל המשימות (לא קל אם משך הזמן בין יצירת תהליכים קצר). כעת נסו לסיים את פעולתם בעזרת התוכנית שכתבתם; זה אמור להיות קל יותר. על מנת להמנע מיצירת תהליכים מרובים מדי, כדאי להריץ את `CloseProcess` לפחות בתחילה עם משך זמן של כ-10 שניות או יותר.

ניהול חבורות תהליכים

מערכות הפעלה מאפשרות בדרך כלל לייצג באופן מפורש קבוצות של תהליכים ולבצע פעולות על כל התהליכים של קבוצה בבת אחת. קבוצה כזו, שנקראת חבורת תהליכים (process group) או עבודה (job) נוצרת באופן מפורש על ידי קריאת מערכת, וניתן לשייך אליה תהליך או תהליכים. כאשר תהליך שמשוייך לחבורה כזו יוצר תהליך נוסף, התהליך החדש משוייך אוטומטית לחבורה (אלא אם משייכים אותו מפורשות לחבורה אחרת). המנגנון הזה גורם לכך שניתן לשייך לחבורה עץ תהליכים שלם, שבו תהליכים יוצרים תהליכים אחרים, גם אם התהליכים הללו אינם משייכים את עצמם לחבורה באופן מפורש. תוכניות מעטפת (shells), למשל, משתמשות במנגנון הזה על מנת לאפשר למשתמש להפסיק את פעולתה של תוכנית שהריץ דרך המעטפת, גם אם התוכנית הזו יצרה מספר גדול של תהליכים, שהמעטפת אינה מודעת כלל לקיומם, ושהתהליך או התהליכים שיצרו אותם אולי כבר אינם קיימים.

בחלונות יש שני סוגים של חבורות תהליכים. סוג אחד נקרא חבורת תהליכים (process group). הסוג הזה דומה לחבורות תהליכים ביוניקס ולינוקס. יוצרים חבורה כזו על ידי ניתוק של תהליך מהחבורה שהוא שייך אליה; זה יוצר חבורה חדשה שהתהליך שנותק שייך אליה וששמה הוא מספר התהליך. הסוג הזה מוגבל למדי, מכיון שניתן להשתמש בו רק כאשר כל התהליכים בחבורה מחוברים לקונסולה אחת. מכיון שתוכניות גרפיות ותוכנות שרת בדרך כלל אינן מחוברות לקונסולה, לא תמיד אפשר להשתמש בסוג הזה.

הסוג השני נקרא עבודה (job), הוא יותר כללי ואפשר לשייך אליו תהליך מכל סוג שהוא. אבל בתרגיל הזה נשתמש בחבורות תהליכים.

יצירת חבורת תהליכים חדשה: חבורת תהליכים חדשה נוצרת כאשר יוצרים תהליך חדש ומעבירים את הדגל CREATE_NEW_PROCESS_GROUP לקריאת המערכת CreateProcess. כל התהליכים שהתהליך החדש ייצור, והצאצאים שלהם, ישויכו אוטומטית לקבוצה הזו אלא אם אחד מהם ייווצר עם אותו הדגל. שם חבורת התהליכים הוא המספר של התהליך שיצרנו עם הדגל CREATE_NEW_PROCESS_GROUP, השורש של עץ התהליכים שמרכיבים את החבורה.

הפסקת ריצה של עבודה שלמה:

```
BOOL GenerateConsoleCtrlEvent(DWORD control_event,
                              DWORD process_group);
```

הקריאה הזו, כאשר ערך הארגומנט הראשון הוא CONTROL_BREAK_EVENT, גורמת לתהליכים של חבורת התהליכים (אולי לא כולם) שמספרה הוא process_group להגיב באותה צורה שהם מגיבים להקשת control-break. בדרך כלל זה גורם להפסקת הריצה של התהליכים שמגיבים לקריאה. התהליכים שיגיבו הם התהליכים ששייכים לחבורה ושמחוברים לאותה קונסולה כמו התהליך שמבצע את הקריאה GenerateConsoleCtrlEvent. תהליכים אחרים בחבורה לא יגיבו.

גם הערך CONTROL_C_EVENT חוקי עבור הארגומנט הראשון, אבל הוא משפיע רק את תהליך בודד שמספרו נתון בארגומנט השני, לא על חבורת תהליכים שלימה.

באמצעות קריאת המערכת SetConsoleCtrlHandler ניתן לקבוע שגרה שתגיב לאירועים הללו, וגם לאירועי סגירה של הקונסולה, יציאה של המשתמש/ת מהמערכת, וכיבוי המערכת. בדרך כלל מטרת שגרת הטיפול היא לדאוג ליציאה מסודרת מהתוכנית על ידי פעולות כמו מחיקת קבצים זמניים ושחרור משאבים אחרים.

התרגיל. עליך לכתוב תוכנית בשם ex-job שניתן להפעיל בשתי דרכים. כאשר מפעילים אותה עם שני ארגומנטים או יותר, היא יוצרת חבורת תהליכים חדשה שהשורש שלה הוא תהליך חדש ששורת הפקודה שלו היא הארגומנטים השני ואילך של ex-pgroup. לפני שהיא יוצרת את התהליך החדש, היא יוצרת אירוע מסוג איפוס ידני ששמו הוא הארגומנט הראשון של ex-pgroup ושמאותחל למצב לא מסומן. לאחר שהתהליך החדש התחיל לרוץ, התוכנית מחכה לאירוע, וכאשר האירוע מסומן, היא מפסיקה את פעולת כל התהליכים שמשווייכים לחבורה.

כאשר מפעילים את התוכנית עם ארגומנט אחד בלבד, היא פותחת את האירוע ששמו הוא הארגומנט הראשון שלה באמצעות קריאת המערכת OpenEvent ומסמנת אותו, מה שאמור לגרום לחבורת התהליכים שהתחלנו אולי קודם לכן להפסיק לרוץ.

הנה הפעלה לדוגמה של התכנית:

↓ the name of the event

```
c:\> ex-pgroup job-event netstat.exe -e 5
```

↑ the command line to run in the new job

כדי להפסיק את פעולת העבודה, מפעילים מתוך חלון אחר את הפקודה

```
c:\> ex-pgroup job-event
```

כדי לבדוק את התוכנית, השתמשו בתוכנית CloneProcess שמסופקת יחד עם התרגיל. צריך להריץ את התוכנית הזו עם ארגומנט אחד, משך זמן בשניות. התוכנית יוצרת שני תהליכים בהפרש של משך הזמן הנתון ואז מסיימת את פעולתה. לאורך זמן, האפקט הוא של יצירת מספר אקספוננציאלי של תהליכים, כאשר התהליכים שיצרו אותם כבר אינם קיימים. צפו ביצירת התהליכים הללו בעזרת מנהל המשימות ונסו לסיים את פעולתם בעזרת מנהל המשימות (לא קל אם משך הזמן בין יצירת תהליכים קצר). כעת נסו לסיים את פעולתם בעזרת התוכנית שכתבתם; זה אמור להיות קל יותר. על מנת להמנע מיצירת תהליכים מרובים מדי, כדאי להריץ את CloseProcess לפחות בתחילה עם משך זמן של כ-10 שניות או יותר.

מיפוי קבצים לזיכרון

בתרגיל זה נלמד למפות קבצים לזיכרון ולהשתמש ביכולת זאת על מנת להעביר נתונים בין שתי תוכניות שרצות בו זמנית.

מיפוי קובץ לזיכרון: מיפוי קובץ לזיכרון מתבצע בחלונות בשני שלבים. בראשון יוצרים עצם מיפוי, שמאפשר למפות חלקים שונים בקובץ לזיכרון, ובשני ממפים איזור מסויים בקובץ (או את כל הקובץ) לזיכרון.

```
HANDLE CreateFileMapping(
           HANDLE  file,
           LPSECURITY_ATTRIBUTES sa,
           DWORD   protection,
           DWORD   max_size_high,
           DWORD   max_size_low
           LPCTSTR mapping_name);
```

הארגומנט הראשון הוא מזהה של קובץ שהתוכנית פתחה קודם לכן, או הערך INVALID_HANDLE_VALUE אם מבקשים למפות לזיכרון חלק מאיזור הדפדוף בדיסק ולא קובץ מסויים. הארגומנט השני מייצג הרשאות שאנו מעוניינים לתת לעצם המיפוי עצמו, או NULL אם אין מעוניינים לציין הרשאות. הארגומנט השלישי מציין האם נרצה לקרוא מתוך הקובץ הממופה לזיכרון, לכתוב לתוכו, או גם לקרוא וגם לכתוב. האפשרויות הללו מיוצגות על ידי הערכים סימבוליים PAGE_READONLY, PAGE_WRITEONLY, ו-PAGE_READWRITE. ניתן גם להעביר את הערך הסימבולי PAGE_WRITECOPY, שמורה למערכת ההפעלה ליצור עותק פרטי באיזור הדפדוף של דפים בקובץ שהתוכנית משנה דרך עצם המיפוי. שני הארגומנטים הבאים מציינים את הגודל המקסימלי של המיפוי. הגודל המקסימלי מיוצג ב-64 סיביות, שמועברות בשני ארגומנטים של 32 סיביות כל אחד. העברת הגודל 0 מציינת שהגודל המקסימלי הוא פשוט גודל הקובץ (אסור להעביר את הגודל 0 אם ממפים את איזור הדפדוף). הארגומנט האחרון הוא שם שאנו מעניקים לעצם המיפוי, אם אנו מעוניינים להשתמש בו מכמה תוכניות.

אם עצם מיפוי בשם הנתון קיים כבר, הקריאה מחזירה מזהה אליו (זו אינה נחשבת שגיאה), אבל במקרה כזה קריאה לפונקציה GetLastError() מחזירה את הערך ERROR_ALREADY_EXISTS.

את עצם המיפוי משחררים, כרגיל, בעזרת CloseHandle. ניתן לקבל מזהה לעצם מיפוי קיים בעזרת הפונקציה OpenFileMapping.

את המיפוי עצמו יוצרים על ידי קריאה לפונקציה הבאה.

```
void* MapViewOfFile(
           HANDLE  file_mapping,
           DWORD   access,
           DWORD   file_offset_high,
           DWORD   file_offset_low
           SIZE_T  view_size);
BOOL UnmapViewOfFile(void* address);
```

הקריאה מקבלת מזהה שהוחזר על ידי CreateFileMapping או OpenFileMappin, ומחזירה מצביע לאיזור בזכרון שאליו מערכת ההפעלה מִיפְתָה את הקובץ. הארגומנט השני מתאר את סוגי הגישה שאנו מעוניינים בהם לקובץ הממופה (הארגומט דומה לארגונמט protection ביצירת עצם המיפוי, אבל משתמש בערכים סימבוליים אחרים):

.FILE_MAP_COPY-ו ,FILE_MAP_ALL_ACCESS ,FILE_MAP_WRITE ,FILE_MAP_READ
שני הארגומנטים הבאים מציינים את ההיסט, הבית הראשון בקובץ שימופה לזכרון, והארגומנט האחרון את גודל השטח שימופה, או 0 אם מעוניינים למפות את כל הקובץ.

ההיסט מתחילת הקובץ חייב להיות כפולה של גודל הדף של מערכת ההפעלה. על מנת לברר את גודל הדף, יש לקרוא תחילה לשגרה GetSystemInfo שמקבלת ארגומנט בודד, כתובת של מבנה מטיפוס SYSTEM_INFO, שהשדה dwPageSize שלו מתאר את גודל הדף בבתים.

מיפוי קבצים היא שיטה יעילה ופשוטה לגישה לקבצים גדולים (פחות העתקות נתונים בתוך מערכת ההפעלה מאשר קריאה וכתיבה רגילה) והיא מאפשרת העברת נתונים בין תהליכים על ידי יצירת עצם מיפוי אחד שמספר תהליכים משתמשים בו.

התרגיל. עליך לכתוב תוכנית בשם win32mmap שניתן להריץ עם ארגומנט של אות בודדת r או w. כאשר המשתמש מריץ את התוכנית פעמיים (בחלונות שונים), כל פעם עם ארגומנט אחר, הוא יכול להקליד טקסט כקלט לתוכנית שהורצה עם הארגומנט w והעותק שהורץ עם הארגומנט r ידפיס את הטקסט כפלט בכל פעם ששורה שלמה הוקלדה. הטקסט שהוקלד יועבר מתהליך הקלט לתהליך הפלט על ידי שימוש במיפוי של איזור הדפדוף. המשתמש גורם לשתי התוכניות לעצור על ידי הקלדה של טקסט שמתחיל בנקודה.

התוכניות מתקשרות ביניהן על ידי מיפוי שטח באיזור הדפדוף לזכרון. התו הראשון באיזור הממופה משמש להעברת פקודות פשוטות בין התוכניות, ושאר השטח משמש להעברת הטקסט. המשמעות של התו הראשון באיזור הממופה היא: הערך נקודה מציין שהמשתמש ביקש לעצור, הערך R מציין שתהליך הקלט כתב מחרוזת באיזור הממופה שעדיין לא נקרא על ידי תהליך הפלט, והערך N מציין שתהליך הפלט הדפיס את כל המחרוזות שתהליך הקלט שלח, והוא ממתין למחרוזות נוספות או לפקודת עצירה. המנגנון הזה להעברת הודעות בדבר מוכנות הזיכרון המשותף לקריאה/כתיבה ולעצירת התוכנית **אינו יעיל ואינו מאפשר מימוש נכון לחלוטין**, אבל הוא מספיק לצורך התרגיל הזה; בהמשך נלמד מנגנונים יעילים ונכונים לתקשורת בין תהליכים.

את הפלט יש להדפיס בעזרת printf ואת הקלט יש לקלוט בעזרת scanf. מותר להניח שאורך שורת קלט מוגבל ל־80 תוים (כלומר מחרוזת של עד 81 תוים כולל ה־0 שמסמן את סוף המחרוזת).

שם העצם המיפוי חייב להתחיל בשם המשתמש שלכם. בעזרת התוכנית WinObj שניתן להוריד מהאתר www.sysinternals.com אפשר לראות את עצם המיפוי תחת BaseNamedObject.

1. כתוב/כיתבי את התכנית. נסו אותה. האם היא עובדת גם כאשר מתחילים קודם את תהליך הפלט וגם כאשר מתחילים קודם את תהליך הפלט (היא צריכה לעבוד בשני המקרים)?

2. האם יתכן שתוכנית הפלט תדפיס פעמיים מחרוזת שהועברה רק פעם אחת על ידי תוכנית הקלט? האם יתכן שתוכנית הפלט תדלג עם מחרוזת? יש לנמק.

3. כעת יש להרחיב את התוכנית כך שאם היא מקבלת יותר מארגומנט אחד, הארגומנט השני הוא שם קובץ שישמש למיפוי, במקום איזור הדפדוף. האם יתכן כעת שתוכנית הפלט תדפיס מחרוזת שלא הוקלדה כלל על ידי המשתמש? האם זה יתכן כאשר משתמשים באיזור הדפדוף? יש לנמק.

יש להגיש את התוכנית לאחר ההרחבה של סעיף 3. יש לודא שהתוכנית משתמשת באיזור הדפדוף אם מעבירים לה רק ארגומנט אחד.

תכנות עם חוטים

בתרגיל זה נלמד לממש תוכנית חלונות שמשתמשת במספר חוטים בו־זמנית. עליכם לממש ספריית שגרות, שצריכות לאפשר לתוכנית הקוראת לבצע קלט ופלט בלי להמתין לסיום הפעולות. בחלונות שגרות הקלט/פלט הרגילות מממשות למעשה ממשק כזה, אך אנו לא נשתמש בו, על מנת לתרגל שימוש בחוטים, מנעולים, ואירועים. המנגנון לקלט/פלט ללא המתנה שמערכות חלונות מספקות דרך הקריאות הרגילות יעיל יותר מהשימוש בחוטים בתרגיל.

יצירת חוט חדש:

```
typedef
  DWORD WINAPI (LPTHREAD_START_ROUTINE*)(void* argu-
ment);
HANDLE CreateThread(
       LPCTSTR security_attributes,
       SIZE_T  stack_size,
       LPTHREAD_START_ROUTINE function,
       LPVOID  function_argument,
       DWORD   creation_flags,
       LPDWORD thread_id);
```

קריאת המערכת הזו יוצרת חוט חדש. החוט החדש מריץ את השגרה funcion, המקבלת מצביע ללא טיפוס (*void) כארגומנט יחיד ומחזירה שלם אי־שלילי. הארגומנט הרביעי של קריאת המערכת הוא המצביע שיועבר כארגומנט ל-funcion. השגרה מחזירה HANDLE, מזהה שמייצג את החוט החדש שנוצר, או NULL אם קריאת המערכת נכשלה.

הארגומנט הראשון של קריאת המערכת מאפשר לציין הרשאות עבור החוט החדש–בתרגיל יש להעביר NULL. הארגומנט השני מציין את גודל המחסנית של החוט החדש בבתים, והערך 0 מורה למערכת ההפעלה להקצות מחסנית בגודל סטנדטי. הארגומנט החמישי מאפשר לשלוט באופן יותר מדוייק על יצירת החוט החדש: הקבוע CREATE_SUSPENED מורה למערכת ההפעלה שהחוט החדש לא יתחיל לרוץ מייד, אלא יווצר כשהוא מושעה, והקבוע STACK_SIZE_PARAM_IS_A_RESERVATION (רק בחלונות xp ואילך) מורה למערכת ההפעלה להקצות מרחב כתובות עבור המחסנית, אך לא להקצות מסגרות פיזיות עד שאלא דרושות. אם יוצרים חוט מושעה, יש לקרוא ל-ResumeThread(HANDLE) על מנת שיתחיל לרוץ. הערך 0 מורה ששני הדגלים הללו אינם בתוקף. הארגומנט האחרון מאפשר למערכת ההפעלה להחזיר מזהה מספרי לחוט החדש. בדרך כלל אין צורך במזהה הזה, וניתן להעביר את הערך NULL, שמורה למערכת ההפעלה שלא להחזיר את המזהה המספרי, אלא רק את ה-HANDLE.

המתנה (כמעט לכל דבר) ונעילת מנעולים. במערכות חלונות יש קריאות מערכת שגורמות לתוכנית לחכות לאירוע כלשהוא, כגון סיום פעולת חוט, לנעילת מנעול, לאירוע, ועוד. בחלונות, כל עצם יכול להמצא באחד משני מצבים: מסומן (signaled) או לא מסומן. חוט או תהליך אינם מסומנים כל זמן שהם רצים או עשויים לרוץ בעתיד, ומסומנים כאשר הם מסיימים את פעולתם. מנעול (mutex) מסומן כאשר הוא נעול ואינו מסומן כאשר הוא חופשי. אירוע (event) הוא פשוט עצם שיכול להמצא באחד משני המצבים הללו, ואין לו משמעות פרט לסימון זה.

קריאת המערכת שנשתמש בה על מנת להמתין שעצם יגיע למצב מסומן היא WaitForSingleObject. הקריאה ממתינה שחוט יסיים את פעולתו, או שמנעול ינעל

על ידי החוט הקורא, או שאירוע (event) מסויים יסומן. אם העצם הוא מנעול, הקריאה חוזרת בהצלחה רק כאשר הוא ננעל על ידי החוט הקורא, וממתינה אם המנעול נעול על ידי חוט אחר. כלומר, על מנעולים קריאת המערכת הזו שקולה לחלוטין ל-lock(m).

```
DWORD WaitForSingleObject(
      HANDLE object,
      DWORD  timeout_in_milliseconds);
```

הארגומנט הראשון הוא המזהה של העצם שמבקשים לחכות לאירוע הקשור בו, והארגומנט השני הוא זמן ההמתנה המקסימלי באלפיות שניה, או הקבוע INFINITE אם ההמתנה אינה מוגבלת בזמן. השגרה מחזירה אחד משלושה ערכים: WAIT_OBJECT_0 אם ההמתנה הסתיימה בהצלחה, WAIT_TIMEOUT אם ההמתנה הסתיימה בגלל שפרק הזמן המקסימלי שציינו עבר, או WAIT_ABANDONED, שמוחזר רק כאשר העצם הוא מנעול (mutex), אם המנעול ננעל בהצלחה משום שהחוט הקודם שנעל את המנעול סיים את פעולתו בלי לשחרר את המנעול. מצב כזה נקרא נטישה של המנעול.

יצירת מנעול (mutex):

```
HANDLE CreateMutex(
       LPSECURITY_ATTRIBUTES security_attributes,
       BOOL    initial_ownership,
       LPCTSTR name);
```

הקריאה יוצרת מנעול חדש, שניתן לתת לו שם גלובלי (הארגומנט השלישי) והרשאות (הארגומנט הראשון). הארגומנט השני מאפשר ליצור מנעול שנעול באופן התחלתי על ידי החוט היוצר. הקריאה מחזירה מזהה או NULL אם היא נכשלת. את המזהה יש לשחרר בעזרת הקריאה CloseHandle(HANDLE). מערכת ההפעלה משחררת את המנעול כאשר לא נשארים מזהים (handles) שמתייחסים אליו. שחרור מנעול נעול מתבצע בעזרת הקריאה ReleaseMutex(HANDLE).

יצירת אירוע (event) ושינוי מצבם:

```
HANDLE CreateEvent(
       LPSECURITY_ATTRIBUTES security_attributes,
       BOOL    manual_reset,
       BOOL    initially_signaled,
       LPCTSTR name);
```

הקריאה יוצרת אירוע חדש, שניתן לתת לו שם גלובלי (הארגומנט הרביעי) והרשאות (הארגומנט הראשון). הארגומנט השלישי מאפשר ליצור מנעול שהמצב ההתחלתי שלו מסומן. הארגומנט השני מציין האם האירוע הוא מסוג איפוס ידני או איפוס אוטומטי. הקריאה מחזירה מזהה או NULL אם היא נכשלת. את המזהה יש לשחרר בעזרת הקריאה CloseHandle(HANDLE). מערכת ההפעלה משחררת את המנעול כאשר לא נשארים מזהים (handles) שמתייחסים אליו.

אירוע שמסומן כאשר יש קבוצת חוטים שממתינה לו מעיר את כולם. אירוע שמסומן כאשר אין ממתינים נשאר מסומן, וחוטים שיקראו לשגרת המתנה לא ימתינו כלל.

את שני סוגי האירועים (איפוס ידני ואוטומטי) מסמנים בעזרת הקריאה SetEvent(HANDLE). אירוע מסוג איפוס ידני מעבירים למצב לא מסומן בעזרת הקריאה ResetEvent(HANDLE). ניתן להעביר גם אירוע מסוג איפוס אוטמטי למצב לא מסומן בעזרת

ResetEvent, אבל אירוע כזה גם עובר למצב לא מסומן ברגע שהוא מעיר חוט אחד או יותר מהמתנה.

בתרגיל זה נממש ספריית שגרות המאפשרת לתכנית לבצע פעולות קלט פלט ללא להמתין, תוך שימוש ב-POSIX threads. ביצוע פעולות בצורה כזו נקרא ביצוע אסינכרוני או nonblocking. הספריה תשתמש בחוטים, כך שפעולת קריאה או כתיבה של התכנית תתבצע למעשה על ידי חוט של הספריה. החוט הוא שימתין לביצוע קריאות המערכת read או write בעוד שהתכנית תוכל להמשיך בפעולות אחרות.

הממשק לספריה שעליכם לממש. עליכם לממש ספריה בת 5 שגרות:

```
int nb_init    (int min_threads, int max_threads);
int nb_finalize();
int nb_read    (HAN-
DLE f, void* buf, SIZE_T count, DWORD offset);
int nb_write   (HAN-
DLE f, void* buf, SIZE_T count, DWORD offset);
int nb_wait    (int request);
```

השגרה nb_init מאתחלת את הספריה. השגרה מאתחלת את מבני הנתונים של הספריה ויוצרת חוטים שיבצעו את פעולות הקלט/פלט עבור לקוחות הספריה. הפרמטר הראשון הוא מספר החוטים שיווצרו בזמן האיתחול, והפרמטר השני הוא המספר המקסימלי שמותר לספריה ליצור. השגרה nb_finalize דואגת לסיום מסודר של כל החוטים של הספריה ומשחררת משאבים שהוקצו על ידי הספריה (כגון זיכרון). השגרות האלו צריכות להחזיר 0 במקרה של הצלחה ו-1- במקרה של כשל.

השגרות nb_read ו-nb_write מתחילות פעולות קריאה או כתיבה. הפעולות חוזרות מייד ללא המתנה, פרט למקרים שיפורטו בהמשך. השגרות מעבירות לספריה מזהה של קובץ פתוח שעליו יש לבצע את הפעולה, מצביע לחוצץ שאליו או ממנו יועברו הנתונים, מספר הבתים שיש להעביר, והמיקום מתחילת הקובץ שממנו או אליו יש להעביר את הנתונים. השגרות מחזירות מזהה פעולה אי־שלילי שמאפשר לתכנית להמתין לסיום הפעולה. במקרה של שגיאה השגרות מחזירות 1-.

השגרה nb_wait ממתינה לסיום פעולת קריאה או כתיבה. הפרמטר הוא מזהה פעולה שהוחזר על ידי nb_read או nb_write.

קריאה ל-nb_read או nb_write מכניסה לתור את פרטי הבקשה. החוטים של הספריה מוציאים בקשות מהתור ומבצעים אותן. מותר להגביל את גודל התור ל-25 בקשות על מנת למנוע בזבוז זיכרון בתור שהולך ומתארך. קריאה ל-nb_read או nb_write שמגלה שהתור מלא ממתינה עד שמתפנה מקום בתור. זה המקרה היחיד שבו קריאות אלה לא יחזרו מייד לתכנית.

מותר גם להגביל את מספר החוטים שהספריה יכולה ליצור ל-25.

תכנית הבדיקה. עליכם לבדוק את הספריה בעזרת תכנית בדיקה בשם tst-win32aio.c שניתן להוריד מהאתר של הספר. התוכנית יוצרת קובץ גדול ולאחר מכן מבצעת עליו עיבודים שונים בשלושה שלבים. השלב הראשון הוא קריאה של כל הקובץ וכתיבתו לדיסק (בבלוקים של 512 KB). השלב השני הוא קריאת כל בלוק בקובץ, טיפול (כבד יחסית) בבלוק בזיכרון, וכתיבתו מחדש. שני השלבים הרשונים יחד מאפשרים להעריך את הזמן שלוקחת הקריאה והכתיבה מהקובץ ואת הזמן שלוקח הטיפול בכל בלוק. בשלב השלישי התכנית קוראת את הבלוקים בצורה אסינכרונית תוך שימוש בספריה שלכם, מטפלת בכל בלוק בזיכרון, וכותבת אותו חזרה בצורה אסינכרונית. בכל איטרציה בשלב זה מתרחשות שלוש פעולות במקביל: קריאה של

הבלוק הבא בקובץ, טיפול בבלוק הנוכחי, וכתיבה של הבלוק הקודם. כל שלב בתכנית מתבצע פעמיים וזמן הריצה שלו מודפס, על מנת שניתן יהיה להעריך בצורה מדוייקת את העלות של כל שלב.

אנו מצפים שעיבוד אסינכרוני יוריד את זמן הריצה של הטיפול בקובץ. נסמן את זמן הריצה הממוצע של השלב הראשון (שרק קורא וכותב לקובץ) ב-t_1, את זמן הריצה של השלב השני ב-t_2, ואת זמן הריצה של השלב השלישי ב-t_3. אנו מעריכים שזמן הריצה הנדרש לטיפול בבלוקים בזיכרון הוא $t_2 - t_1$. זמן הריצה של השלב השלישי חייב לכן לקיים $t_3 \geq \max\{t_2 - t_1, t_1\}$. אנו מקווים ש-$t_3 \approx \max\{t_2 - t_1, t_1\}$, אך בפועל זמן הריצה יהיה לרוב גדול יותר. בכל מקרה שבו $t_3 < t_2$ נדע שזמן הריצה ירד הודות לביצע קלט/פלט בצורה אסינכרונית. הנה פלט אופייני של תכנית הבדיקה, שמראה שזמן הריצה אכן יורד, אך החישוב בזיכרון לא מתבצע במקביל לחלוטין לקלט/פלט:

```
Y:\Courses\os\exercises\win32aio\Release>win32aio 100 1 4 start-
ing worker 0
temporary file name=<c:\temp\delete.me>
writing 100 MB (200 ios of 512 KB each)
Time = 6.109 seconds
Syncronous Processing, no computation
Time = 14.718 seconds
Syncronous Processing,
no computation Time = 14.546 seconds
Syncronous Processing, with computation (x1024)
flag = 1 (ignore)
Time = 27.499 seconds
Syncronous Processing, with computation (x1024)
flag = 1 (ignore)
Time = 27.718 seconds
Asyncronous Processing, with computation (x1024)
creating another worker (1)
creating another worker (2)
flag = 1 (ignore)
Time = 15.046 seconds
Asyncronous Processing, with computation (x1024)
flag = 1 (ignore)
Time = 14.968 seconds
NB Worker <0>: stopping
NB Worker <2>: stopping
NB Worker <1>: stopping
```

כאן $t_1 \approx 14.5$, $t_2 \approx 25.5$, ו-$t_3 \approx 15.0$ (בשניות). כלומר $t_3 < t_2$ אבל $t_3 > \max\{t_2 - t_1, t_1\}$.

תכנית הבדיקה מקבלת שלושה ארגומנטים מספריים: גודל הקובץ ב-MB ומספר החוטים המינימלי והמקסימלי.

ציון המקום בקובץ שממנו צריך לקרוא או לכתוב. הארגומנט החמישי של קריאות המערכת ReadFile ו-WriteFile הוא מצביע למבנה מטיפוס OVERLAPPED שמכיל חמישה איברים:

```
typedef struct {
  ULONG_PTR Internal, InternalHigh;
  DWORD     Offset, OffsetHigh;
  HANDLE    event;
} OVERLAPPED;
```

שני השדות הראשונים הם לשימוש המערכת. שני השדות הבאים מציינים את 32 הסיביות התחתונות ו-32 הסיביות העליונות של המיקום. הספריה שלנו תאפשר לגשת לקבצים בגודל 4 GB בלבד, כך שהסיביות העליונות יהיו תמיד 0. השדה האחרון מאפשר להעביר מזהה של אירוע שמערכת ההפעלה תעביר למצב מסומן כאשר פעולת הקלט או הפלט תסתיים–אנו לא נשתמש במנגנון הזה ולכן יש למלא את השדה הזה בערך 0.

אנו נפתח את הקובץ **בלי** הדגל FILE_FLAG_OVERLAPPED. העברת מצביע למבנה OVERLAPPED בפעולה על קובץ שנפתח ללא הדגל הזה גורמת לפעולת הקלט/פלט להתבצע מהמיקום המצויין בשדות ה-Offset, וקריאות הקלט/פלט יחזרו לחוט הקורא רק כאשר הפעולות יסתיימו. הדגל הזה מאפשר לבצע קלט/פלט ללא המתנה ללא שימוש בחוטים, אך כאמור אנו לא נשתמש במנגנון זה.

התרגיל. כתוב/כיתבי ספריה בשם ex-aio-win32.c המממשת את הספריה כפי שהוגדרה.

ניתן לממש את התרגיל בשתי רמות. ברמת המימוש הפשוטה יש ליצור את מספר החוטים המינימלי שמצוין בקריאה ל-nb_init ותו לא. ברמת המימוש המלאה יש ליצור מספר זה של חוטים ב-nb_init, אך קריאה ל-nb_read או nb_write שמגלה שכל החוטים הקיימים עסוקים ושנוצרו פחות חוטים מהמספר המקסימלי שצויין בקריאה ל-nb_init, חייבת ליצור חוט נוסף שיבצע קלט/פלט.

בכל מקרה יש לממש תור בקשות. הקריאות nb_read ו-nb_write צריכות להכניס את הבקשה לתור ולאותת במידת הצורך שאינו ריק. חוטי הקלט/פלט של הספריה צריכים למשוך בקשות מהתור ולבצע אותן. כאשר מוצאת בקשה מהתור יש לאותת, במידת הצורך, שניתן להכניס בקשות נוספות לתור.

אסור לבצע דגימה (polling) לשום צורך שהוא. כל ההמתנות חייבות לבקש זמן המתנה אינסופי.

יש להריץ את תכנית הבדיקה תוך שימוש בספריה שלכם על קובץ גדול. על מנת לקבל תוצאות משמעותיות, הריצו את גרסת ה-Release ולא את גרסת ה-Debug. כדאי להתבונן בחיווויי הביצועים במנהל המשימות בזמן ריצת התוכנית. יש לצרף לתכנית המוגשת, בהערות, את הפלט של התכנית בהרצות עם מינימום ומקסימום של חוט אחד ועם מינימום ומקסימום של ארבעה חוטים. למי שמגיש את רמת המימוש המלאה, יש להגיש גם פלט עם מינימום ומקסימום של אפס חוטים ומקסימום של ארבעה.

קריאה ממדריכים וסיווג קבצים

בתרגיל זה נלמד לקרוא תוכן מדריכים ולבחון תכונות של קבצים בלינוקס ויוניקס.

קריאת תוכן מדריך:

```
#include <windows.h>
...
HANDLE FindFirstFile(LPCTSTR pattern,
                     WIN32_FIND_DATA* find_data);
BOOL   FindNextFile (HANDLE search_handle,
                     WIN32_FIND_DATA* find_data);
BOOL   FindClose    (HANDLE search_handle);
```

קריאת המערכת FindFirstFile מתחילה חיפוש של קבצים במדריך. הארגומנט הראשון שלה הוא תבנית של שמות קבצים שמבקשים לחפש. התבנית כוללת תחילית של שם מדריך וסיומת של תבנית של שם קובץ, שיכולה להכיל את התוים * ו-?, כמו למשל C:\Temp*.doc, C:*, או *\home-fs\stoledo\\. יש לשים לב לכך שהמחרוזת צריכה תמיד להסתיים בתבנית של שמות קבצים, כך שאם היא מסתיימת בשם שם מדריך, ללא סיומת כגון *\, החיפוש ימצא אך ורק קובץ אחד, המדריך עצמו, ולא את כל הקבצים שבמדריך. ארגומנט שמצביע על מדריך השורש של אות כונן, כמו C:, אינו חוקי. הארגומנט השני הוא מצביע למבנה שבו מוחזר מידע אודות קובץ אחד שנמצא בחיפוש. השגרה מחזירה מזהה שבעזרתו ניתן לקבל מידע אודות שאר הקבצים שנמצאו בחיפוש, ויש לסגור את המזהה הזה על ידי קריאה ל-FindClose.

קריאת המערכת FindNextFile מחזירה מידע על קובץ נוסף שעונה לתנאי החיפוש. הארגומנט הראשון שלה הוא המזהה שהוחזר על ידי FindFirstFile, והשני הוא מצביע למבנה שיכיל מידע אודות הקובץ. לקריאה הזו קוראים בדרך כלל בלולאה, למציאת כל הקבצים שעונים לחיפוש. כאשר אין יותר קבצים שעונים לתנאי החיפוש, הקריאה נכשלת והערך שיוחזר מ-GetLastError() הוא ERROR_NO_MORE_FILES.

מידע אודות קובץ: מנגנון חיפוש הקבצים מחזיר מידע אודות קובץ או מדריך במבנה מסוג WIN32_FIND_DATA,

```
typedef struct {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[MAX_PATH];
  TCHAR    cAlternateFileName[14];
} WIN32_FIND_DATA;
```

השדה הראשון מכיל מספר ביטים שכל אחד מהם מייצג תכונה אחרת של הקובץ, כגון FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_DIRECTORY, ו-FILE_ATTRIBUTE_REPARSE_POINT, (על משמעות התכונה האחרונה נעמוד בהמשך). שלושת השדות הבאים מייצגים את הזמן שבו הקובץ נוצר, שבו נגשו אליו לאחרונה, ושבו כתבו אליו לאחרונה. תוכנית יכולה לשנות את השדות הללו של קובץ, כך שהמידע אינו בהכרח אמין. ניתן לחלץ מהשדות הללו ייצוג קריא (ב-UTC, השעה הסטנדרטית בגריניץ') בעזרת השגרה FileTimeToSystemTime. שני השדות הבאים מייצגים את גודל הקובץ בבתים בעזרת שני שדות של 32 סיביות כל אחד. יש לשים לב שגודל הקובץ יכול להיות גדול מכדי שניתן יהיה לייצג אותו בעזרת משתנה של 32 סיביות, כמו int. הנוסחה שבעזרתה ממירים את שני השדות הללו לגודל קובץ היא (nFileSizeHigh * (MAXDWORD+1)) + nFileSizeLow. כאמור, יש להיזהר מגלישה בחישוב הזה (למשל על ידי שימוש במשתנים מסוג double או ULARGE_INTEGER). שם הקובץ שנמצא מופיע בשדה cFileName, ואם יש לו שם נרדף קצר (8 תווים לכל היותר לפני הנקודה ועוד 3 לכל היותר אחריה), הוא יופיע בשדה האחרון.

יצירת וספירת מצביעים לקבצים: מערכת הקבצים NTFS תומכת ביצירת מצביעים נוספים לקבצים. המצביעים הללו הם למעשה שמות נרדפים לקובץ. ניתן ליצור שם נרדף לקובץ במדריך אחר מזה שבו הקובץ שוכן (אבל רק באותה מערכת קבצים), וניתן להעביר כל מצביע ממקום למקום, או לשנות את שמו, באופן בלתי תלוי. בין השמות הנרדפים של קובץ אין סדר של בכירות: ניתן למשל למחוק את המצביע המקורי של הקובץ אבל הקובץ ימשיך להישמר במערכת הקבצים תחת השם הנרדף. ניתן ליצור מצביע כזה בעזרת הפקודה fsutil create hardlink בתוכנת מעטפת (cmd.exe). הריצו את הפקודה בלי ארגומנטים נוספים על מנת לקבל פירוט של משמעות הארגומנטים הנוספים. ניתן גם ליצור מצביע בעזרת קריאת המערכת CreateHardLink. לא ניתן ליצור מצביע למדריך, על מנת שלא יווצרו מעגלים במרחב השמות.

את מספר המצביעים לקובץ ניתן לברר באמצעות קריאת המערכת הבאה.

```
#include <windows.h>
...
BOOL GetFileInformationByHandle(HANDLE hFile,
        BY_HANDLE_FILE_INFORMATION* finfo);
```

הקריאה מקבלת מזהה של קובץ פתוח ומחזירה מידע אודות הקובץ במבנה שכתובתו מועברת בארגומנט השני. המבנה הזה דומה למדי ל-WIN32_FIND_DATA, פרט לשדה nNumberOfLinks שמתאר את מספר המצביעים לקובץ, ושלושה שדות אחרים שמאפשרים לדעת האם שני מזהים פתוחים (HANDLE's) מתייחסים לאותו קובץ, אולי דרך מצביעים אחרים. (יש לשים לב שבניגוד למידע שמוחזר על ידי FindFirstFile ו-FindNextFile, שעבורו אין צורך לפתוח את הקובץ, על מנת לקבל את המידע הזה צריך לפתוח את הקובץ).

מצביעים סימבוליים ונקודות הצבה: מערכת הקבצים NTFS תומכת בעוד שני סוגים של מצביעים. שני הסוגים הללו משתמשים במנגנון שנקרא reparse point. המנגנון הזה מאפשר לשנות את ההתנהגות של מערכת ההפעלה בזמן שתוכנית פותחת קובץ או מדריך מסויים. המנגנון פועל על ידי הוספת רשומת מידע לקובץ או למדריך. הרשומה כוללת את סוג המידע ואת המידע עצמו. כאשר פותחים את הקובץ/מדריך, מערכת ההפעלה מזהה את המידע הנוסף, מבררת את סוגו, ומפעילה שגרה של מערכת ההפעלה שהיא מיוחדת לאותו סוג מידע. המנגנון הזה מאפשר להוסיף למערכת ההפעלה יכולות כגון העברה של קבצים שהשימוש בהם נדיר לספריות רובוטיות של קלטות או דיסקים נשלפים, למשל.

מצביעים סימבוליים ונקודות הצבה הם שני סוגים של reparse points. נקודת הצבה מאפשרת להציב מערכת קבצים שלמה על גבי מדריך (שצריך להיות קיים וריק). זה מאפשר להשתמש במערכת הקבצים כאילו היא הייתה חלק מממערכת קבצים אחרת, ולא דרך אות כונן. התוכנית `mountvol` מאפשרת ליצור נקודות הצבה ולהסיר אותן.

מצביע סימבולי, שנקרא בחלונות צומת (junction) הוא מדריך דמה שה-reparse point שלו גורמת למערכת ההפעלה לפתוח מדריך אחר. זהו למעשה שם נרדף למדריך. בניגוד למצביעים שתיארנו קודם, יש יחס בכירות בין המדריך עצמו ובין מצביע סימבולי אליו. למשל, ניתן למחוק מדריך שיש אליו מצביע סימבולי. במקרה כזה המצביע הסימבולי פשוט יפסיק לעבוד. מיקרוסופט לא מספקת ביחד עם מערכת ההפעלה כלי שמאפשר ליצור ולמחוק מצביעים סימבוליים, אבל ניתן לנהל מצביעים כאלה בעזרת התוכנית `junction` שניתן להוריד חינם מ-www.sysinternals.com ובעזרת התוכנית `linkd` שהיא חלק מה-resource kit של חלונות (חבילת תוכנה של מיקרוסופט שאינה מופצת יחד עם חלונות). למעשה, מערכת ההפעלה מתייחסת לנקודות הצבה ולמצביעים סימבוליים כאילו היו בדיוק אותו סוג של reparse point.

על מנת למצוא את סוג ה-reparse point ששייך למדריך ועל מנת לקרוא את המידע ששמור בו (כולל שם המדריך או מערכת הקבצים שמצביע סימבולי או נקודת הצבה מצביעים אליהם), צריך להשתמש בקריאת המערכת הבאה.

```
#define _WIN32_WINNT 0x0500 /* Windows 2000 and up */
#include <windows.h>
#include <winioctl.h>
#define FSCTL_GET_REPARSE_POINT 0x000900a8
typedef struct {
    DWORD  ReparseTag;
    WORD   ReparseDataLength;
    WORD   Reserved;
    union {
        struct {
            WORD   SubstituteNameOffset;
            WORD   SubstituteNameLength;
            WORD   PrintNameOffset;
            WORD   PrintNameLength;

WCHAR  PathBuffer[1]; /* can be larger! */
        } MountPointReparseBuffer;
        struct {
            BYTE   DataBuffer[1];
        } GenericReparseBuffer;
    };
} MY_REPARSE_DATA_BUFFER
...
```

```
BOOL DeviceIoControl(HANDLE file,
                     DWORD  command,
                     void*  input_buffer,
                     DWORD  input_buffer_size,
                     void*  output_buffer,
                     DWORD  output_buffer_size,
                     DWORD* bytes_returned,
                     OVERLAPPED* overlapped);
```

לקריאת המערכת הזו צריך להעביר מזהה של קובץ פתוח ואת הפקודה FSCTL_GET_REPARSE_POINT. לקריאה הזו הרבה שימושים, כמו למשל יצירת reparse point. בשימוש שלנו, אין צורך בקלט נוסף, כך שהארגומנט השלישי יכול להיות NULL והרביעי 0. בשני הארגומנטים הבאים יש להעביר שטח זיכרון שבו יכתב המידע שב-reparse point ואת אורכו. אורך המידע עשוי להגיע ל-16384 בתים, ולכן יש להעביר שטח זכרון בגודל מספיק.

אם ה-reparse point הוא מסוג מצביע סימבולי או נקודת הצבה, אז השדה ReparseTag במבנה שיוחזר יכיל את הערך IO_REPARSE_TAG_MOUNT_POINT. במקרה כזה השדות SubstituteNameOffset ו-SubsituteNameLength יכילו את המיקום והאורך של מה שמוצב במדריך הדמה (מיקום כהיסט בבתים מתחילת השדה PathBuffer). השם של מה שמוצב מקודד תמיד ב-Unicode, ולכן יש לוודא שהתוכנית היא תוכנית Unicode או לפחות שמתייחסים למחרוזת הזו באופן מתאים.

כאשר פותחים קובץ או מדריך מתוך כוונה להעביר את במזהה שלו לקריאת המערכת הזו על מנת לקרוא reparse point, יש לפתוח את הקובץ עם הדגלים FILE_FLAG_OPEN_REPARSE_POINT ו-FILE_FLAG_BACKUP_SEMANTICS זה גורם לפתיחה של קובץ או מדריך הדמה שה-reparse point מוצמד אליו, ולא לפתיחה של מה שמצביעים עליו.

ההגדרות של FSCTL_GET_REPARSE_POINT ושל MY_REPARSE_DATA_BUFFER נחוצות משום שסביבת הפיתוח של Win32 לא תמיד מגדירה אותן.

התרגיל.

1. צרו נקודת הצבה עבור כונן התקליטורים וודאו שהיא פועלת.

2. צרו מדריך f ובו שני קבצים, a ו-b. כעת צרו בהורה של המדריך מצביע c (hard link) ל-b ומחקו את b מ-f. האם תוכנו עדיין זמין מ-c?

3. צרו מדריך g בהורה של f וצרו אליו מצביע סימבולי gj ב-f. האם אתם יכולים ליצור מעגל על ידי יצירת מצביע fj ל-f בתוך g? מה קורה כאשר מחפשים בקבצים במדריכים הללו (דרך תוכנת חיפוש הקבצים של מערכת ההפעלה)? מה קורה אם מוחקים את המדריך g?

4. כעת כתבו תוכנית בשם ex-win32dir שמדווחת על תוכן מדריך, בדומה לפקודה dir. עבור כל קובץ במדריך, התוכנית צריכה להדפיס את זמן הכתיבה האחרונה לקובץ, תו שמציין האם הקובץ הוא מדריך או לא, את מספר המצביעים אל הקובץ, את גודלו, את שמו הקצר (אם יש), את שמו הארוך, ואם הוא נקודת הצבה או מצביע סימבולי, את מה שהוא מצביע עליו. הפלט בהמשך מדגים את פעולת התוכנית הדרושה. יש להגיש, בנוסף לתוכנית ולתשובות על השאלות הקודמות, פלט של התוכנית שלכם על מדריך שיש בו קובץ עם יותר ממצביע אחד, מדריך עם נקודת הצבה, ומדריך עם מצביע סימבולי (מותר שזה יהיה אותו מדריך).

```
C:\os\exercises\ex-win32dir\Debug> ex-win32dir .
22/04/2004 09:00 D (1)      0           .
22/04/2004 09:00 D (1)      0           ..
21/04/2004 12:09 - (1)     17 THI-
SIS~1  This is a long name
21/04/2004 12:09 - (2)     17           b
21/04/2004 12:09 D (1)      0           gj -> ..\g
...
```

דפדפן זעיר

מטרת תרגיל זה היא לכתוב תכנית שמורידה קובץ משרת אינטרנט (ליתר דיוק, משרת HTTP). פרוטוקול HTTP הוא פרוטוקול להעברת קבצים שתוכנן במיוחד על מנת לקשר בין דפדפנים לשרתי אינטרנט. הפרוטוקול משתמש בפרוטוקול העברה מסוג TCP, פרוטוקול להעברת נתונים בקשר סדור ואמין.

איתחול סביבת התקשורת: בחלונות צריך לאתחל ולסגור את סביבת התקשורת,

```
#include <winsock2.h>
...
int WSAStartup(WORD version, WSAData* data);
int WSACleanup();
int WSAGetLastError();
```

בהשגרה הראשונה מאתחלת את סביבת התקשורת, השנייה סוגרת אותה, והשלישית מחזירה את קוד השגיאה האחרון, שניתן להעביר ל-FormatMessage על מנת לקבל מחרוזת שמתארת את השגיאה. הארגומנט הראשון של שגרת האיתחול מאפשר לתוכנית לוודא שהגרסה של שגרות התקשורת עדכנית; השתמשו בערך MAKEWORD(2,2). כמו כן, בסביבת הפיתוח יש לוודא שהתוכנית משתמשת בספריה wsock32.lib (בדרך כלל תחת project properties > linker > input > additional dependencies).

יצירת וסגירת שקע תקשורת:

```
#include <winsock2.h>
...
SOCKET socket(int domain, int type, int protocol);
int    closesocket(SOCKET s);
```

הקריאה socket יוצרת שקע ומחזירה מזהה מספרי (כמו open עבור קבצים) או 1- (בחלונות הקבוע INVALID_SOCKET) במקרה של כשל. הפרמטר הראשון מציין את "עולם התקשורת" שבו יפעל השקע. לתקשורת בפרוטוקולי האינטרנט ערכו צריך להיות הקבוע הסימבולי AF_INET. הארגומנט השני מציין את סוג השירות שהשקע יספק. ליצירת קשר סדור ואמין (למשל קשר TCP) יש להעביר את הערך SOCK_STREAM. הארגומנט השלישי מציין את הפרוטוקול הספיציפי ועבור TCP ערכו צריך להיות IPPROTO_TCP. את השקע סוגרים בעזרת קריאת המערכת close או closesocket.

חיבור שקע קיים לשרת:

```
#include <winsock2.h>
...
int connect(SOCKET           sockfd,
            struct sockaddr* server_name,
            int              server_name_len);
```

הקריאה מחברת את השקע לשרת. במקרה של כשל מוחזר הערך 1-. הארגומנט הראשון הוא מספר השקע שהוחזר בקריאה ל-socket. הארגומנטים השני והשלישי מציינים את השרת

שאליו רוצים להתחבר ואת מספר השער (port). איתחול מבנה זה מוסבר בהמשך. הארגומנט השלישי מציין את אורך הארגומנט השני.

מרגע שהשקע מחובר לשרת ניתן לקרוא ולכתוב ממנו ואליו על ידי שימוש ב-read ו-write ממש כמו מקובץ או צינור (מספר השקע משמש כמזהה של קובץ פתוח). על מנת לסגור את הקשר יש לקרוא ל-close, ממש כמו לקובץ פתוח.

בניית כתובת אינטרנט. המבנה מסוג struct sockaddr הוא מבנה פולימורפי שיכול ליצג כתובות במשפחות פרוטוקולים שונות. במקרה שלנו יש להשתמש במבנה שמייצג כתובות אינטרנט. שם המבנה הספיציפי הזה הוא struct sockaddr_in (יש לבצע casting למצביע בזמן הקריאה ל-connect). במבנה זה שלושה שדות: (1) sin_family, המציין את סוג הכתובת ובמקרה של כתובות אינטרנט עליו להכיל את הקבוע AF_INET, (2) sin_port, מטיפוס short, שצריך להכיל את מספר השער כאשר הבתים מסודרים ב-network order, (3) sin_addr שמכיל את כתובת השרת, כפי שהיא מוחזרת מהקריאה gethostbyname או gethostbyaddr שיתוארו בהמשך. השגרה htons ממירה משתנה מטיפוס short מסידור בתים של המחשב לסידור הקאנוני של הרשת (מוגדרת ב-netinet/in.h).

את ארגומנט אורך המבנה בקריאה ל-connect (הארגומנט השלישי) יש לחשב בעזרת sizeof של struct sockaddr_in.

```
#include <winsock2.h>
...
struct hostent* gethostbyname(const char *name);
struct hostent* gethost-
byaddr(const char *addr, int len, int type);
```

שתי השגרות הללו הופכות מחרוזת המכילה שם שרת או את כתובת ה-IP שלו למשתנה שניתן להעתיק (בעזרת memcpy) לשדה sin_addr במבנה מסוג struct sockaddr_in. שתי השגרות מחזירות NULL אם הן נכשלות. במקום לנסות לגלות האם המשתמש העביר לתוכנית שם שרת או כתובת מקובל פשוט לקרוא לשגרה הראשונה ואם היא נכשלת לקרוא לשניה. הארגומט הראשון בשתי השגרות הוא המחרוזת המכילה שם או כתובת שרת. הארגומנט השני ל-gethostbyaddr הוא פשוט אורך המחרוזת והארגומנט השלישי צריך להיות AF_INET.

השגרות מחזירות כתובת של מבנה שמכיל כתובת (או מספר כתובות אם לשרת יש מספר כתובות). אורך כל כתובת מצויין בשדה h_length. האורך הוא מספר הבתים שיש להעתיק לחבר sin_addr במבנה מסוג struct sockaddr_in. הכתובות עצמן שמורות בשדה שהוא מערך של מצביעים לכתובות ששמו h_addr_list. ניתן פשוט להשתמש בכתובת הראשונה במערך.

פרוטוקול HTTP. על מנת לקבל שירות משרת, הלקוח מתחבר בקשר TCP לשרת, בדרך כלל לשער מספר 80, ושולח בקשה. הבקשה מכילה מספר שורות ולאחריהן שורה ריקה המסמנת את סוף הבקשה. אנו נשתמש במבנה בקשה פשוט הכולל שלוש שורות. השורה הראשונה מכילה פקודה. אנו נשתמש בפקודה GET. באותה שורה, אחרי הפקודה, יופיע מזהה הקובץ שרוצים להוריד (ה-URL), ולאחריו מזהה הפרוטוקול, מופרדים ברווחים. אנחנו נשתמש בגרסה הראשונה של פרוטוקול HTTP שהמזהה שלו הוא HTTP/1.0. השורה השנייה תכיל את התג Host: ולאחריו (מופרד ברווח) שם שרת שמסוגל לספק את הקובץ המבוקש. זה לא חייב להיות השרת שהתחברנו אליו או השרת שמצוין ב-URL, אם כי השרת המצויין ב-URL הוא השרת ההגיוני ביותר לשדה זה. השורה השלישית תהיה שורה ריקה. כל שורה צריכה להסתיים בתוים \r\n (ולא \n בלבד). הנה בקשה לדוגמא:

```
GET http://www.math.tau.ac.il/~sivan/ HTTP/1.0\r\n
Host: www.tau.ac.il\r\n
\r\n
```

השרת עונה במספר שורות מידע לגבי השרת והקובץ ולאחריהן שורה ריקה, ולאחריה הקובץ (או תחילתו במקרה של קובץ גדול). שורת המידע הראשונה היא החשובה ביותר מכיוון שהיא מכילה אינדיקציה להצלחת או כשלון השירות. השורה המכילה את גודל הקובץ שיישלח גם היא חשובה. להלן דוגמא אופיינית לשורות המידע כפי שהתקבלו כתגובה לבקשה שהצגנו:

```
HTTP/1.0 200 OK
Date: Tue, 25 Apr 2000 09:50:57 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Wed, 22 Mar 2000 13:44:01 GMT
ETag: "cf999-1007-38d8ce21"
Accept-Ranges: bytes
Content-Length: 4103
Content-Type: text/html
```

את האפיון המלא של פרוטוקול HTTP ניתן למצוא באתר www.w3.org.

התרגיל. עליך לכתוב תכנית בשם ex-browse שתפקידה לקבל קובץ משרת HTTP ולהדפיס אותו ואת שורות המידע ששולח השרת לפלט. לתכנית יש ארבעה ארגומנטים: שם/כתובת שרת שאליו יש להתחבר, מספר שער, שם השרת שיספק את הקובץ, וה-URL של הקובץ עצמו. למשל, ההתחברות בדוגמה היא תוצאה של הפקודה

```
% ex-browse www.tau.ac.il 80 \
  www.math.tau.ac.il \
  http://www.math.tau.ac.il/~sivan/
```

הסבר קצר לגבי שרתים מורשים (proxy servers): ספקי אינטרנט רבים (כולל אוניברסיטות וחברות שמספקות שירותי אינטרנט למחשבים בתוך הארגון) חוסמים התחברויות בערוצי TCP לשער מספר 80 של שרתים חיצוניים, וזאת על מנת ליעל את תעבורת ה-HTTP. על מנת להוריד קבצים כאשר המחשב מחובר לספק אינטרנט כזה, יש לבקש את הקבצים משרת מורשה (proxy) ששייך לספק ושרק לו יש יכולת ליצור חיבורים חיצוניים לשער 80. בדרך כלל, השליח מבקש עבורנו את הקובץ מהשרת החיצוני. אולם אם יש לשליח עותק עדכני של הקובץ מכיון שלקוח כלשהו הוריד אותו לאחרונה, הוא יחזיר לנו את העותק ששמור אצלו, ובכך יחסוך לספק האינטרנט תעבורת IP. ספק האינטרנט או אנשי התמיכה באוניברסיטה או בחברה יוכלו לומר לכם את שם השרת המורשה, אם נעשה שימוש בשרת כזה.

ניתן ללמוד רבות על התנהגות שרתי אינטרנט מניסויים עם הדפדפן הזעיר, למשל:

1. אם הספק שלכם משתמש בשרת מורשה, נסה/נסי להוריד את דף הבית של אתר כמו www.cnn.com בבקשה ישירה ל-www.cnn.com וגם דרך השרת המורשה. באיזה מהדרכים הצלחת להוריד את הדף? אם נתקלתם בכשל, מה בדיוק קרה?

2. נסו להוריד את דף האינטרנט של הספר, למשל, עם וגם בלי "/" אחרי המילה osbook. באחד המקרים תקבלו ככל הנראה הודעת שגיאה. מה היתה הודעת השגיאה? איך מתנהג דפדפן מקצועי כאשר הוא מקבל הודעת שגיאה כזו (נסו)?

שרת אינטרנט לתוכן דינמי

מטרת תרגיל זה היא לכתוב שרת HTTP פשוט שיהיה מסוגל לספק תוכן דינמי. כלומר, השרת אינו מספק לדפדפן תוכן של קבצים, אלא הוא מריץ תוכניות לפי בקשת הדפדפן ושולח חזרה לדפדפן את הפלט שלהן. השרת מיועד למערכות חלונות, אם כי מבחינת קריאות המערכת הקשורות לתקשורת, תוכנות שרת בלינוקס/יוניקס ממומשות בדיוק באותה צורה. התרגיל משלב למעשה התנהגות של שרת קבצים והתנהגות של מעטפת (shell).

האזנה לבקשות התחברות לשער מסויים. יצירת קשר TCP בין שרת ולקוח, למשל דפדפן, אינה פעולה סימטרית. בתרגיל הקודם ראינו שהלקוח מתחבר לשרת על ידי קריאה ל-connect לאחר יצירת השקע. השרת, לעומת זאת, ממתין לבקשות התחברות. ראשית, השרת צריך להכין את השקע:

1. השרת יכול לשנות את התנהגות השקע על ידי קריאה ל-setsockopt אחרי יצירתו. אין חובה לעשות זאת, אך ברירות המחדל של התנהגות שקע מתאימות ללקוחות ואינן מתאימות בדיוק לשרתים. שני פרמטרים שרצוי לשנות הם: (1) להרשות שימוש חוזר מיידי במספרי שער, (2) להשאיר את הקשר חי עד שכל הנתונים נשלחו בהצלחה. בתרגיל זה אין צורך לשנות את התנהגות השקע.

2. השרת קושר את השקע לשם מסוים שאליו יתחברו הלקוחות על ידי קריאה ל-bind. השם הוא בדרך כלל כתובת ה-IP שדרכה רוצים לקבל בקשות התחברות (לשרת יכולות להיות מספר כתובות אם יש לו מספר חיבורים לרשת) ומספר שער שמוסכם על השרת והלקוחות כאחד. שרתי HTTP משתמשים בשער מספר 80 כברירת מחדל. (גם לקוח יכול לקרוא ל-bind לפני הקריאה ל-connect אם הוא רוצה מספר שער מסוים עבור הקשר, אך בדרך כלל אין סיבה לקשור את שקע של לקוח למספר שער מסוים).

3. השרת מודיע על רצונו ביצירת תור של לקוחות שמבקשים להתחבר על ידי קריאה ל-listen. בקריאה השרת מודיע מה צריך להיות אורך התור המקסימלי. אם יותר לקוחות מנסים להתחבר, בקשתם תידחה. תור ארוך שימושי למצבים שבהם השרת עלול להיות עמוס עד כדי שאינו מסוגל לפתוח ערוצים בקצב קבלת הבקשות.

לאחר הכנת השקע, ההמתנה עצמה מתבצעת על ידי קריאה ל-accept. קריאה זו מכניסה את השרת (או לפחות את החוט או התהליך שקרא ל-accept) להמתנה עד שלקוח מנסה להתחבר. כאשר לקוח מנסה להתחבר, הקריאה חוזרת ומחזירה מזהה שקע חדש. מזהה זה מייצג את הקשר הפתוח לתקשורת וניתן לקרוא ולכתוב ממנו בעזרת read ו-write. בסיום השימוש בקשר הזה, יש לסגור אותו בעזרת close. השקע המקורי שעליו ביצענו accept אינו משמש לתקשורת עם הלקוח וניתן לקרוא איתו שוב ל-accept על מנת לאפשר ללקוחות נוספים להתחבר. בדרך כלל, שרתים משתמשים בחוט/תהליך אחד לביצוע accept בלולאה אינסופית, ובחוט/תהליך נפרד לטיפול בכל קשר שנפתח ללקוח.

```
#include <winsock2.h>
...
int bind  (SOCKET socket, struct sock-
addr* my_addr,     int  addr_len);
int listen(SOCKET socket, int queue_size);
int accept(SOCKET socket, struct sock-
addr* client_addr, int* addr_len);
```

הארגומנטים השני והשלישי ל-bind מציינים את כתובת השרת כולל מספר השער שאליו רוצים לקשור את השקע. בונים אותם כמו בדיוק כמו שבונים כתובת ל-connect. את שם המחשב שעליו רץ השרת ניתן לקבל על ידי קריאה ל-gethostname.

הארגומנט השני ל-listen הוא מספר בקשות ההתחברות שיכולות להמתין בתור לפני שבקשות ידחו.

הארגומנטים השני והשלישי ל-accept משמשים את השגרה להחזרת כתובת הלקוח. ניתן גם לקבל את כתובת הלקוח על ידי קריאה לשגרה getpeername לאחר יצירת הקשר. ניתן להפוך את הכתובת למחרוזת על ידי שימוש ב-inet_ntoa.

טיפול בקשר ללקוח. כפי שהוסבר כבר, בדרך כלל רצוי בשרת לקרוא ל-accept שוב מייד לאחר שקריאה קודמת מחזירה מזהה של שקע שמחובר לקשר פתוח, ולהשאיר את הטיפול בקשרים הפתוחים לתהליכים או חוטים אחרים. אנו בונים שרתים בצורה זאת על מנת למנוע מצב שבו לקוח ממתין זמן רב להתחברות משום שהטיפול בלקוח קודם לוקח זמן רב.

אנו נשתמש בתרגיל זה באסטרטגיה פשוטה לטיפול בלקוחות. כל פעם ש-accept יוצרת קשר חדש, השרת יקרא ל-CreateProcess על מנת ליצור תהליך חדש שיטפל בקשר שנוצר. התהליך המקורי צריך פשוט לסגור את השקע החדש שנוצר (הוא לא ישתמש בו) ולקרוא שוב ל-accept. התהליך החדש שנוצר צריך לסגור את השקע שעליו בוצע accept, לטפל בבקשת השירות מהלקוח, ואז לסגור את הקשר ולצאת.

אסטרטגיה זו אינה יעילה במיוחד מכיון שהיא יוצרת תהליך חדש, פעולה יקרה, בכל בקשת התחברות. עדיף היה להשתמש במאגר של תהליכים או חוטים שאינם מתים לאחר טיפול בלקוח אחד, כפי שעשינו תרגיל הקלט/פלט ללא המתנה.

כאשר יש צורך לטפל במספר גדול של לקוחות בו־זמנית גם אסרטגיה זו עלולה להיות לא יעילה מפאת המחיר של מיתוג תכוף של תהליכים או חוטים. ניתן להתמודד עם בעיה זו על ידי שימוש בקריאת המערכת select שמאפשרת לתהליך בודד לטפל במספר גדול של ערוצים בו זמנית. התרגיל בפרק הבא מציג את הגישה הזו.

כאמור, אנו נשתמש באסטרטגיה של יצירת תהליך חדש עבור כל בקשת שירות. התהליך החדש שנוצר צריך להיות מסוגל להשתמש בשקע הפתוח ללקוח, על מנת לקבל ממנו את הבקשה ולשלוח לו את התשובה. את היכולת הזו צריך להעניק לו על ידי הורשת המזהה של השקע לתהליך החדש. הדרך הפשוטה ביותר להוריש לו את המזהה היא לשכפל את המזהה, כך שהמזהה החדש, שמתייחס לאותו שקע, יהיה בר ירושה, ולבקש בזמן יצירת התהליך החדש שהוא יירש מזהים (רק את המזהים שנוצרו באופן מפורש כברי ירושה).

```
#include <windows.h>
...
BOOL DuplicateHandle(
       HANDLE  source_process,
       HANDLE  source_handle,
       HANDLE  target_process,
       HANDLE* target_handle,
       DWORD   desired_access,
       BOOL    inheritable_target,
       DWORD   options);
```

קריאת המערכת הזו משכפלת מזהה, ויכולה גם להעביר אותו מתהליך אחד לתהליך אחר. אנו נשתמש בה על מנת לשכפל מזהה בתוך התהליך הנוכחי, כך שנעביר בארגומנט הראשון והשלישי את הערך החוזר מ-GetCurrentProcess(). בארגומנט השני מעבירים את המזהה שרוצים לשכפל, וברביעי את כתובת המשתנה שיקבל את המזהה המשוכפל. אנו

רוצים שהמזהה החדש ייהיה בר הורשה, ולכן נעביר את הערך TRUE. בארגומנט השישי. הארגונמנטים החמישי והשביעי מאפשרים לשלוט בהרשאות הגישה שהמזהה המשוכפל יספק. מכיון שאנו רוצים את אותן הרשאות כמו במזהה המשוכפל, נעביר בארגומנט האחרון את הערך DUPLICATE_SAME_ACCESS, ואז הקריאה תתעלם מהארגומנט החמישי (העבירו 0).

כדי שהתהליך החדש יוכל להשתמש במזהה המשוכפל שירש, צריך להעביר את הערך TRUE. בארגומנט החמישי של CreateProcess(), שמודיע למערכת ההפעלה שהתהליך החדש צריך לרשת את המזהים ברי ההורשה של התהליך הנוכחי. כמו כן, התהליך החדש צריך לדעת מהו המזהה, מה מספרו. בתרגיל הזה נעביר לו את מספר המזהה בשורת הפקודה.

הרצת תכנית עבור לקוח ואיסוף הפלט. על השרת להריץ תכנית עבור הלקוח ולשלוח לו את הפלט שלה. מכיון שעל השרת לשלוח לפני הפלט את אורך הפלט (בשורת המידע Content-Length), על השרת לאסוף את הפלט בחוצץ, לספור את אורכו ורק לאחר מכן לשלוח אותו ללקוח עם שורות מידע מתאימות.

איסוף הפלט מתבצע בעזרת צינור חסר שם שהשרת יוצר בעזרת קריאת המערכת CreatePipe.

```
#include <windows.h>
...
BOOL CreatePipe(HANDLE* read_pipe,
                HANDLE* write_pipe,
                SECURITY_ATTRIBUTES* sa,
                DWORD   size);
```

הקריאה מחזירה שני מזהים שניתן לכתוב לאחד מהם ולקרוא מהשני. הארגונמט האחרון הוא בגדר הצעה למערכת בנוגע לגודל החוצץ שצריך להקצות עבור הצינור, אבל הוא אינו מגביל בשום אופן את כמות המידע שניתן להעביר בצינור. את המזהה שניתן לכתוב אליו צריך לשכפל כך שניתן יהיה להוריש אותו לתוכנית שנריץ, כדי שהיא תשתמש בו בתור ערוץ הפלט הסטנדטי שלה. לאחר מכן השרת קורא ל-CreateProcess על מנת ליצור תהליך שיריץ את התכנית שהלקוח ביקש (בדומה למה שעשינו בתרגיל שנושאו הרצת תהליכים ותכניות). כדי שהתהליך החדש ישתמש בצינור בתור ערוץ הפלט, צריך לשנות במבנה מסוג STARTUPINFO- שאת כתובתו מעבירים ל-CreateProcess (ארגומנט לפני אחרון) שני שדות: בשדה dwFlags צריך להדליק את הקבוע STARTF_USESTDHANDLES, ובשדה hStdOutput צריך לשמור את המזהה שאליו התהליך החדש צריך לכתוב את הפלט שלו.

```
ZeroMemory( &start_info, sizeof(STARTUPINFO) );
start_info.cb         = sizeof(STARTUPINFO);
start_info.dwFlags    = STARTF_USESTDHANDLES;
start_info.hStdOutput = write_pipe_duplicate;
```

לאחר יצירת התהליך החדש יש לאסוף את הפלט שלו לתוך חוצץ, לחכות לסיומו, למדוד את כמות המידע בחוצץ, ולשלוח את המידע ללקוח. כעת התהליך שטיפל בלקוח יכול לסיים את פעולתו.

התרגיל. עליך לכתוב תכנית בשם win32-serve שתתפקד כשרת HTTP המריץ תכניות ושולח את פלטן ללקוח. לתכנית יש ארגומנט אחד, מספר שער שיש להאזין לו (הריצו אותו עם מספר גבוה, המספרים עד 1023 שמורים ל-root).

השרת מפרש את ה-URL שהלקוח שלח בבקשת ה-GET כשם תכנית וארגומנטים בצורה הבאה: שם הקובץ הוא שם תכנית שיש להריץ, ובצמוד אליו, מופרדים בסימני + מופיעים הארגומנטים לתכנית. למשל, ה-URL

```
http://localhost:2000/fsutil+volume+diskfree+c:
```

יפורש על ידי שרת שרץ באותו מחשב ומאזין לשער מספר 2000 בתור בקשה להריץ את התכנית `fsutil` (במדריך שבו רץ השרת) עם הארגומנט הנתונים. השרת מחפש את שם התכנית והארגומנטים בתוך ה-URL על ידי חיפוש ה-/ האחרון ב-URL והפרדת המחרוזת משם ואילך בעזרת סימני +.

השרת צריך להחזיר header בן שלוש שורות, שורת רווח, ולאחריה הפלט מהתכנית. כל שורה ב-header צריכה להסתיים ב-`\r\n`. שלוש שורות ה-header צריכות להיות

```
HTTP/1.0 200 OK
Content-Length: 133
Content-Type: text/plain
```

השורה הראשונה מציינת קוד הצלחה, השניה את אורך קובץ הפלט שישלח (והיא צריכה כמובן להכיל את האורך האמיתי של הפלט, לא את הקבוע 339), והשלישית את סוג הקובץ, כאן טקסט פשוט. אין צורך להחזיר קוד שגיאה מדויק במקרה של כשל (למשל אם התכנית אינה קיימת ו-`execv` נכשל).

השרת צריך להגיב רק לבקשות מסוג GET מלקוחות. על השרת לקרוא בכל מקרה את כל בקשת ה-HTTP עד לשורת הרווח המציינת את סיומה (כלומר עד רצף של שתי מחרוזות `\r\n` רצופים).

על מנת לבדוק את השרת שלכם, כדאי להעתיק למדריך שבו אתם מריצים אותו תכנית כגון `date.exe` או `fsutil.exe` מתוך המדריך `windows\system32`.

השרת צריך להדפיס לתוך הפלט שלו את כל בקשות ה-HTTP שהוא מקבל. (מבקשות אלה ניתן ללמוד על מבנה הבקשות שדפדפנים שולחים).

מפאת מורכבות התרגיל, מוצע לממש אותו בשלושה שלבים ולבדוק את פעולתו לאחר כל שלב:

1. בשלב ראשון ממשו שרת שמתעלם מה-URL ותמיד שולח שורת פלט קבועה ללקוח (בצירוף header מתאים כמובן). כמו כן, השרת משרת את הלקוח ללא יצירת תהליך חדש. השרת משרת בקשה אחת, סוגר את הקשר, וקורא שוב ל-`accept`.

2. כעת צרו תהליך חדש עבור כל לקוח.

3. בשלב השלישי הוסיפו את פענוח ה-URL והרצת התכנית המבוקשת לפי הנדרש בתרגיל.

ניסויים שיש לערוך עם השרת:

1. הריצו את השרת וגשו אליו מדפדפן כגון mozilla או internet explorer, רצוי ממחשב אחר.

2. הריצו את התכנית `date` בעזרת השרת וודאו שאתם מקבלים תאריך מעודכן כל פעם שאתם לוחצים על כפתור ה-reload או ה-refresh של הדפדפן.

3. אם יש לכם גישה לשרת מורשה, נסו לגשת לשרת שלכם ישירות (לא דרך השרת המורשה) וגם דרך השרת המורשה והשוו את בקשות ה-HTTP שהשרת מקבל.

הרשאות ורשימות הרשה/סרב

מטרת תרגיל זה היא לכתוב תוכנית שתוכל להציג את רשימת ההרשאות של קובץ, לקבוע רשימת הרשאות מוגדרת לקובץ, ולהעתיק הרשאות מקובץ אחד לאחר.

קריאת רשימת ההרשאות של עצם. קריאת המערכת GetSecurityInfo קוראת את רשימת ההרשאות של עצם כגון קובץ, מנעול, אירוע, תהליך, וכדומה. בחלונות כמעט כל עצם במערכת ההפעלה הוא בר הגנה על ידי רשימת הרשאות. הקריאה מחזירה מבנה מסוג SECURITY_DESCRIPTOR שיש לו ארבעה שדות: המשתמש/ת בעל/ת העצם (owner), קבוצת המשתמשים (group) שאליה שייך העצם, רשימת הרשאות (discretionary access control list, או בקיצור dacl), ורשימת בקרת גישה (system access control list או sacl). ניתן לבקש מקריאת המערכת כל תת קבוצה של ארבעת השדות הללו; הקריאה לא מחזירה בהכרח את כולם.

```
#include <aclapi.h>
...
DWORD GetNamedSecurityInfo(
        TCHAR*                name,
        SE_OBJECT_TYPE        object_type,
        SECURITY_INFORMATION  SecurityInfo,
        SID**                 owner,
        SID**                 group,
        ACL**                 dacl,
        ACL**                 sacl,
        SECURITY_DESCRIPTOR** security_descriptor);
```

הארגומנט הראשון הוא שם העצם, למשל שם קובץ, והארגומנט השני הוא סוג העצם. עבור קבצים, ערך הארגומנט השני צריך להיות SE_FILE_OBJECT. מערכת ההפעלה תומכת בקריאה נוספת, GetSecurityInfo שבה מזהים את העצם על ידי מזהה פתוח, ולא על ידי שמו.

הארגומנט השלישי מציין את השדות שמבקשים לקרוא, והוא צריך להכיל איחוד (or) של תת־קבוצה של ארבעת הקבועים

- DACL_SECURITY_INFORMATION
- OWNER_SECURITY_INFORMATION
- GROUP_SECURITY_INFORMATION
- SACL_SECURITY_INFORMATION

הקריאה מחזירה את המבנה שנקרא בעזרת הארגומנט האחרון. ליתר דיוק, הקריאה קובעת ערך של מצביע למבנה, שכתובתו (של המצביע) מועברת בארגומנט האחרון. לאחר השימוש במידע יש לשחרר את המבנה בעזרת קריאת המערכת LocalFree. בנוסף למצביע הזה, הקריאה קובעת את ערכם של עד ארבעה מצביעים נוספים לחלקים של המבנה. יש להעביר כתובות של מצביעים עבור כל סוג מידע שרוצים לקרוא. אם לא מבקשים, בארגומנט השלישי, סוג מידע מסויים, ניתן להעביר NULL במקום מצביע למצביע מתאים בארגומנטים 7-4.

כדי לקרוא את רשימת בקרת הגישה, את זהות בעליו של העצם, ואת זהות הקבוצה שלה שייך העצם, צריך הרשאה מסוג READ_CONTROL לעצם. כלומר, בהחלט יתכן שאסור יהיה לקרוא את המידע הזה. כדי לקרוא את רשימת רישום הגישה, התהליך הקורא צריך להיות בעל זכות מיוחדת (privilege). בתרגיל זה אין צורך לקרוא את הרשימה הזו.

פענוח זהות המשתמש והקבוצה. הרשאה מתירה או אוסרת למישהו פעולה מסויימת. ישות כזו, שניתן להרשות לה ולאסור עליה פעולות, נקראת מורשה. בחלונות לא רק משתמשים בודדים הם מורשים, אלא גם קבוצות שרירותיות של משתמשים, מחשבים הם מורשים, ועוד. מורשה יכול להיות מוגדר במחשב מסויים, או במרחב שמות שנקרא תחום (domain). תחום מכיל בדרך כלל את שמות המורשים באירגון מסויים או בחלק של אירגון. קריאת המערכת הקודמת שתיארנו מחזירה את זהות בעל העצם והקבוצה לא כשמות, אלא כמזהים של מורשים. את המזהים צריך לפענח לשמות על מנת להציגם.

```
#include <windows.h>
...
BOOL LookupAccountSid(
       TCHAR*         system_name,
       SID*           sid,
       TCHAR*         name,
       DWORD*         name_length,
       TCHAR*         domain,
       DWORD*         domain_length,
       SID_NAME_USE*  what_kind);
```

קריאת המערכת מקבלת, בארגומנט השני, מצביע למזהה של מורשה, ומתרגמת אותו למחרוזת. התרגום מתבצע במחשב ששמו מועבר בארגומנט הראשון למחרוזת, או במחשב שעליו רצה כעת התוכנית, אם בארגומנט הראשון מועבר NULL.

שם המורשה, שמורכב משם תחום (domain name) ושם המורשה, מוחזר בארגומנטים החמישי והשלישי. הארגומנטים הרביעי והשישי מתארים לקריאת המערכת את גודל החוצצים שהוקצו לשמות הללו. אם אחד החוצצים או שניהם קטנים מידי, הקריאה נכשלת וגודל החוצץ הדרוש מוחזר במשתנים שכתובתם הועברה.

הארגומנט האחרון מחזיר את סוג המורשה, שיכול להיות, בין היתר, אחד מהערכים הבאים

- `SidTypeUser`
- `SidTypeGroup`
- `SidTypeWellKnownGroup`
- `SidTypeComputer`
- ועוד סוגים נוספים, כגון תחום וחשבון שנסגר.

פענוח רשימת בקרת הגישה. רשימת בקרת הגישה (dacl) מורכבת מסידרה של איברי בקרת גישה (access control entries או ace). מספר האיברים שמור בשדה `AceCount` של מבנה מטיפוס ACL. קריאת המערכת `GetAce` קוראת איבר ברשימה, על פי אינדקס שמתחיל מ-0, ומחזירה אותו במבנה מסוג `ACCESS_ACE`.

```
#include <windows.h>
...
BOOL GetAce(ACL*   acl,
            DWORD  index,
            VOID** ace);
```

הקריאה מחזירה איבר ברשימה על ידי קביעת ערך מצביע אליו. איברים מסוגים שונים מיוצגים על ידי מבנים מטיפוסים שונים. על מנת שניתן יהיה לזהות את טיפוס המבנה המוחזר, המבנה מכיל תמיד תחילית קבועה שמתארת, בין היתר, את סוג האיבר. התחילית היא מטיפוס ACE_HEADER והשדה AceType של התחילית מתאר את סוג האיבר. הסוגים שאנו נטפל בהם הם

- ACCESS_ALLOWED_ACCESS_TYPE
- ACCESS_DENIED_ACCESS_TYPE

אבל יש עוד סוגים רבים אחרים. במקרה של איבר הרשאה (הסוג הראשון שציינו) המבנה המוחזר הוא מסוג ACCESS_ALLOWED_OBJECT_ACE. שני השדות החשובים במבנה זה הם Mask שמתאר את הפעולות המותרות (זהו שלם שבו כל סיבית מציינת פעולה מותרת אחת), ו-SidStart, שכתובתו היא כתובת ה-SID של המורשה שהאיבר מתאר הרשאות שלו. כלומר, יש לחשב את כתובת השדה הזה, ולהשתמש בה בתור כתובת של מבנה מטיפוס SID. הסיבה לממשק המוזר היא ככל הנראה שהמבנה הזה הוא בעל גודל משתנה. איבר איסור גישה בנוי בצורה דומה, תוך שימוש במבנה מטיפוס ACCESS_DENIED_OBJECT_ACE. כדאי להשתמש ב-union על מנת להשתמש במצביע ש-GetAce מחזיר בצורות שונות, תחילית ולאחר מכן מבנה מסוג מסויים.

הפעולות המותרות או האסורות מתוארות על ידי סיביות דלוקות בשדה Mask. ניתן לבחון סיביות ספיציפיות (ולהדליק ולכבות אותן) תוך שימוש בקבועים הבאים (שלושת הראשונים ספיציפיים לקבצים, והשאר שימושיים לכל סוג עצם):

- FILE_GENERIC_READ
- FILE_GENERIC_WRITE
- FILE_GENERIC_EXECUTE
- GENERIC_READ
- GENERIC_WRITE
- GENERIC_EXECUTE
- GENERIC_ALL
- DELETE
- READ_CONTROL
- WRITE_DAC
- WRITE_OWNER
- SYNCHRONIZE

יצירת רשימת בקרת גישה חדשה. כעת נראה כיצד ליצור רשימת בקרת גישה חדשה. את הרשימה יש ליצור בשטח זיכרון רצוף שאותו מקצים בעזרת LocalAlloc. בהמשך נראה כיצד לחשב את הגודל הנדרש. קודם כל, נלמד כיצד להכין את רשימת בקרת הגישה.

```
#include <windows.h>
...
BOOL InitializeAcl(ACL*  acl,
                   DWORD acl_length,
                   DWORD acl_revision_level);
BOOL AddAccessAllowedAce(
                   ACL*  acl,
                   DWORD acl_revision,
                   DWORD access_mask,
                   SID*  sid);
BOOL AddAccessDeniedAce(כנ"ל);
```

הקריאה הראשונה מאתחלת את הרשימה. צריך להעביר לה מצביע לשטח הזיכרון שהוקצה, את גודל השטח, ואת הגירסה של הרשימה הדרושה (יש שתי גרסאות לרשימות הללו, פשוטה ומתוחכמת יותר). הגרסה יכולה להיות אחד מבין שני הערכים ACL_REVISION ו-ACL_REVISION_DS. הגרסה השנייה מאפשרת שימוש בהרשאות שהן ספיציפיות לעצמים מסוגים שונים, כגון תהליכים, קבצים, וכדומה. בגרסה הראשונה, שמספיקה לצורך התרגיל הזה (ושהיא היחידה שנתמכת במערכות הפעלה ישנות), ניתן לייצג רק פעולות גנריות.

שתי הקריאות הנוספות מוסיפות כל אחת איבר אחד לרשימה, איבר התרה או איבר איסור. בשתיהן הארגומנט הראשון הוא מצביע לרשימה, השני הוא גרסה (כמו בשגרת האיתחול), השלישי הוא הוא איחוד (or) של הפעולות המותרות או האסורות, והרביעי הוא מזהה של מורשה.

סדר האיברים ברשימה החדשה הוא סדר הוספתם.

גודל השטח שמוקצה לרשימה צריך להיות שווה לגודל מבנה מסוג ACL ועוד סכום גודלי האיברים. האיברים הם בעלי גודל משתנה, בגלל שגודל ה־SID משתנה, ולכן צריך לחשב את גודל כל איבר בעזרת נוסחה כגון

```
   sizeof(ACCESS_DENIED_ACE)-
sizeof(DWORD)+GetLengthSid(sid);
```

קריאת המערכת GetLengthSid מחזירה כמובן את גודל המזהה. הסיבה לחיסור גודל מילה היא שהשדה SidStart מסמן את תחילת המזהה, אבל הוא למעשה חלק מהמזהה.

קישור הרשימה שיצרנו לעצם מתבצע על ידי קריאת המערכת הבאה

```
#include <aclapi.h>
...
DWORD SetNamedSecurityInfo(
        TCHAR*                name,
        SE_OBJECT_TYPE        object_type,
        SECURITY_INFORMATION  SecurityInfo,
        SID*                  owner,
        SID*                  group,
        ACL*                  dacl,
        ACL*                  sacl);
```

השימוש בקריאה הזו דומה לשימוש בקריאה GetNamedSecurityInfo, פרט לכך שכאן מעבירים מצביעים למבנים ולא מצביעים למצביעים.

התרגיל. עליך לכתוב תכנית בשם win32-acl שתוכל להדפיס את רשימת בקרת הגישה ושם הבעלים של קובץ, וכן לקבוע רשימה חדשה לקובץ קיים. כאשר מריצים את התוכנית עם ארגומנט אחד, שם קובץ, היא מדפיסה את שם הבעלים של הקובץ (משתמש וקבוצה), ואת רשימת בקרת הגישה של הקובץ. למשל,

```
c:> win32-acl c\:temp
owner: [U] SWIFT\Administrator
group: [G] SWIFT\None
Allow: [A] BUILTIN\Administrators
generic:---- file:RWE
...
Allow: [W] \CREATOR OWNER
generic:A--- file:---
...
```

האות שמופיעה בסוגריים מרובעים לפני כל מורשה מתארת את סוג המורשה, U למשתמשים, G לקבוצות שרירותיות, ו-W לקבוצות "ידועות", כלומר מוגדרות באופן סתום ולא מפורש. התוכנית צריכה לתאר, כמו בדוגמה, את סוג כל איבר ברשימה (הרשה או סרב), את המורשה, ואת ההרשאות הגנריות והרשאות הקבצים, כאשר כל אות מתארת סוג הרשאה.

אם מפעילים את התוכנית עם שני ארגומנטים, שהראשון מביניהם הוא שם קובץ והשני הוא סימן קריאה, התוכנית צריכה לקשר את הקובץ לרשימה חדשה בת שני איברים. האיבר הראשון צריך להרשות לבעל הקובץ את כל סוגי הגישה הגנרית, כולל קריאה ושינוי הרשאות ובעלות, והאיבר השני צריך לאסור על בעל הקובץ כתיבה לתוכו.

בנוסף למימוש התרגיל, יש לענות על השאלות הבאות.

1. צרפו לתשובתכם את פלט התוכנית על הקובץ c:\temp ועל קובץ הריצה של התוכנית.
2. צרו קובץ חדש בעורך טקסטים כלשהו, וצרפו את פלט התוכנית על קובץ הטקסט הזה.
3. כעת שנו את הרשאות הגישה לקובץ הטקסט באמצעות התוכנית שלכם, ונסו לקרוא ולכתוב ממנו ואליו. האם הצלחתם? צרפו את הפלט שמתאר את ההרשאות עכשיו.
4. עכשיו יש לנסות לבחון את ההרשאות בעזרת הממשק הגרפי של חלונות. קליק ימני על הקובץ, מאפיינים, security. מה קורה, ולמה לדעתכם זה קורה?
5. הרשו לחלונות לשנות את ההרשאות על פי המוצע בדיאלוג. צרפו פלט התוכנית שלכם כעת על אותו קובץ, ונסו לקרוא ולכתוב. האם זה מותר?