מערכות הפעלה / סיכום תרגולים ותרגילי בית

סיכום system-calls חשובות:

System call: return value	Parameter	Usage
WriteConsoleOutputCharacter: BOOL	מקבלת handle ל-console, מחרוזת, אורך המחרוזת, קורדינטות לקונסול ומצביע לערך בו יוחזר מס' התווים שנכתבו. פותחת את הקונסול לפי הקורדינטות וכותבת בו את המחרוזת.	
CreateFile: HANDLE	• _in LPCTSTR lpFileName, - שם • _in DWORD dwDesiredAccess, - ההרשאות שפותח הקובץ דורש (קריאה או כתיבה או שניהם) • _in DWORD dwShareMode,- ההרשאות שפותח הקובץ מאפשר לאחרים (למשל קריאה או כתיבה) • _in_opt ... lpSecurityAttributes, - לא חשוב • _in DWORD dwCreationDisposition, - מה לעשות אם הקובץ קיים/לא קיים (למשל: צור תמיד, פתח רק אם קיים וכו') • _in DWORD dwFlagsAndAttributes, - למשל: o FILE_FLAG_Sequential_scan: הקובץ יקרא רצוף o FILE_FLAG_Random_access: הקובץ יקרא באופן לא רציף • _in_opt HANDLE hTemplateFile - לא חשוב	פתיחת קובץ, מחזירה handle לקובץ.
SetFilePointer: DWORD	• _in HANDLE hFile – מזהה לקובץ • _in LONG lDistanceToMove – מספר הבתים לזוז • _inout_opt PLONG lpDistanceToMoveHigh, - לא חשוב • _in DWORD dwMoveMethod – מאיפה לזוז: FILE_BEGIN, FILE_CURRENT, FILE_END	הזזת המצביע הפנימי של הקובץ (עבור אותו handle בלבד).

Handles:

Handle הוא מזהה לאובייקטים, כאשר לכל תהליך יש רשימת handles. ה-handle הוא כניסה לטבלת ה- handles של תהליך, ולכן חסר משמעות מכל תהליך אחר. כניסה מכילה את כתובת מבנה הנתונים של המערכת לאותו תהליך, וככזה הוא מוגן. יכולים להיות כמה handles לאותו אובייקט. המערכת מחזיקה handle count ו-reference count. המערכת משחררת משאב של אובייקט אם אין לו יותר handles. אובייקט מחזיק:

- HandleCount: מס' ה-handles לאובייקט מכלל התהליכים.
- ReferenceCount: מס' ההצבעות הכולל עליו. למשל, אובייקט event שמצביע עליו handle מתהליך שמשתמש בו, וגם thread כלשהו שממתין לו – סה"כ 2 הצבעות.

קבצים:

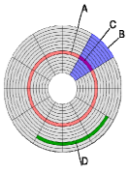
- ReadFile: מקבל handle לקובץ, מצביע לבפר ומספר bytes לקרוא, וקורא מהקובץ (מהמיקום הנוכחי של ה-file-pointer שלו) לבפר.
- WriteFile: אותו עקרון, רק מקבל בפר ומספר bytes שצריך לכתוב לקובץ.

שני ה-bytes הראשונים בקובץ מסמלים האם הוא ב- unicode או ב-ASCII. ניתן לבדוק את הסיגנטורה ע"י קריאתם (קריאת WORD = 2 בתים), והשוואה ל-UNICODE_SIGNATURE. האופציה (_T("<some_string>" מאפשרת כתיבה בהתאם (Unicode/ascii).

- FindFirstFile:HANDLE: מקבלת מחרוזת חיפוש (כולל * כמשתנה כללי) לקובץ/תיקיה, שומר את תוצאת החיפוש הר אשונה בתוך מבנה מסוג WIN32_FIND_DATA, ממנו אפשר אח"כ להוציא פרטים על הקובץ שנמצא. מחזיר handle **לחיפוש**.
- FindNextFile:HANDLE: אותו עקרון, ממשיך את החיפוש, רק מקבל כפרמטר ראשון את ה-handle שהוחזר מהחיפוש הקודם (מזהה ל**חיפוש**, לא לקובץ, זה שהוחזר ע"י findFirst או findNext). שומר ב-WIN32_FIND_DATA את נתוני הקובץ החדש, מחזיר handle מעודכן לחיפוש.
- FindClose:BOOL: סוגר את ה-handle לחיפוש. הערה לשניים לעיל: מחזירים INVALID_HANDLE_VALUE אם נגמר החיפוש.
- CloseHandle:BOOL: סוגר handle, למשל מזהה לקובץ פתוח.
- GetFileSize:DWORD: מקבל handle לקובץ ומחזיר את גודלו ב-bytes.

Cluster היא יחידת הזיכרון הקטנה ביותר לשימוש, וקבצים שמורים על clusters שלמים. אם סוף קובץ תופס רק חלק מה-cluster, חלקו האחרון לא תפוס ע"י אחר. גודל ה-cluster נקבע ע"י מערכת הקבצים (למשל NTFS/FAT32). קריאות מערכת שקשורות לכך:

- GetDiskFreeSpace: מקבלת כקלט את ה-path של הדיסק הרצוי ומחזירה בתוך פרמטרים שונים מידע על: מספר הסקטורים בקלסטר, מספר בתים בסקטור, מספר קלסטרים פנויים, מספר הקלסטרים הכולל. חישוב גודל קובץ: מס' קלסטריםXמס' סקטור בקלסטרXמס' בתים בסקטור.



A: track , מסילה.
B : geometrical sector, כל הסקטורים על אותה זוית.
C : track sector, הסקטור על אותה מסילה.
D : cluster, מספר הסקטורים הנקראים כיחידה אחת, בהתאם למערכת הקבצים של הדיסק.

<u>סיכום תרגיל בית 1 : Dir2File</u> :

- חישוב פונ' hash על קובץ : קיבלה path לקובץ, פתחה handle לקובץ (לקריאה, שיתוף קריאה, לפתוח אם קיים בלבד, random access), קריאת הבית הראשון (לא צריך לשחק עם ה-file-pointer כי הוא בהתחלה), כיוון מצביע הקובץ לסוף פחות 1, קריאת הבית האחרון, חישוב ה-hash.
- חישוב גודל קלסטר לתיקיית הקלט : שימוש ב-tsplitpath_ על התיקיה כדי להוציא ממנה את שם הכונן. שימוש ב-getDiskFreeSpace על הכונן, החזרת bytes_per_sector X sectors_per_cluster.
- Main : יצירת קובץ output (לכתיבה, אסור כלום לאחרים, יצירה תמיד, כתיבה רציפה), כתיבת 2 בתים של Unicode_sign לקובץ (כדי שיהיה בפורמט Unicode). לאחר מכן מתחיל החיפוש על התיקיה משורשר ל- "*.*" (אתחול משתנה מסוג WIN32_FIND_DATA ע"י ZeroMemory), כאשר כל עוד התיקיה לא ריקה: ע"י השוואת flags בודקים שהקובץ אינו קובץ system או תת-תיקיה, ואז מחשבים את גודלו:

$$file - size - ondisk = cluster - per - file \times cluster - size(bytes)$$

בכתיבת התוכן (שנכתב למחרוזת בינתיים) לקובץ הפלט, פרמטר המספר בתים לכתוב הוא גודל המחרוזת 2X, כי כל תו תופס כפליים (כי זה Unicode). בסוף הלולאה בודקים עם GetLastError() שהוחזר ERROR_NO_MORE_FILES, ולא, למשל, שהחיפוש נכשל. סוגרים handles.

סיכום system-calls חשובות :

System call: return value	Parameter	Usage
CreateProcess:BOOL	• __in_opt LPCTSTR lpApplicationName, - למשל: _T("program.exe") • __inout_opt LPTSTR lpCommandLine, - למשל: _T("program.exe 1") • __in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes, - NULL • __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes, - NULL • __in BOOL bInheritHandles, - לא חשוב • __in DWORD dwCreationFlags, - CREATE_NO_WINDOW (המנע מליצור חלון) • ... (פרמטרים לא חשובים) • __out LPPROCESS_INFORMATION lpProcessInformation – מצביע למבנה נתונים שמחזיק מידע על התהליך	יצירת תהליך. ה-handle לתהליך יימצא ב-process information, לא יוחזר ע"י הפונקציה. שם גם ימצא handle ל-thread הראשי (ברירת המחדל).
WaitForSingleObject:DWORD	• __in HANDLE hHandle, - מזהה לאובייקט שרוצים לחכות עד שיאותת • __in DWORD dwMilliseconds – time-out	חזרות חשובות : WAIT_OBJECT_0 – האובייקט signaled, WAIT_TIMEOUT – עבר זמן ההמתנה שהוגדר בקריאה.
CreateThread:HANLDE	• __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes, - לא חשוב, NULL • __in SIZE_T dwStackSize, - 0 , גודל מחסנית התחלתי • __in LPTHREAD_START_ROUTINE lpStartAddress, - פונקציה שתרוץ בחוט זה • __in_opt LPVOID lpParameter, - העברת פרמטרים לחוט • __in DWORD dwCreationFlags, • __out_opt LPDWORD lpThreadId	יצירת חוט מפונקציה תוך העברת פרמטרים וערכים אחרים (פחות חשובים). מחזיר handle לחוט החדש.

<u>ניהול multi-tasking : processes</u> :

מערכת Windows מנהלת משימות שרצות בניהול יעיל והוגן , מנהלת את המצבים של המשימה , שומרת על הזיכרון הפרטי שלה ממשימות אחרות ומנהלת את ניהול השימוש שלה במשאבים . משימה יכולה להיכנס למצב ממתין /חוסם שימנע מהמערכת להקצות לה זמן מעבד . משימה מבוטאת ב-process : אובייקט שיש לו handle, אינסטנציה של תוכנית הרצה על המחשב.

מאפיינים של תהליך :

- מזהה : handle.
- מצב : running, waiting...
- עדיפות : גבוהה, נמוכה.
- PC : כתובת בזכרון לפקודה הבאה.
- מצביעי זכרון : מיקום זכרון התהליך בזכרון הפיסי.
- Context data : רגיסטרים.
- Accounting info : זמן מעבד ששומש ע"י התהליך עד כה.

- GetExitCodeProcess : מחזיר את קוד היציאה של תהליך, למשל:
- TerminateProcess : מקבלת handle לתהליך ומצביע לקוד יציאה, מסיימת את התהליך ושמה בפרמטר השני את קוד היציאה . אם התהליך טרם התסיים, יוחזר STILL_ACTIVE.

מה-process information, מבנה הנתונים שמתקבל ביצירת התהליך , ניתן להוציא מידע על : זמן יצירה, זמן יציאה, זמן kernel וזמן user. בכל תהליך נוצר לו גם חוט ולכן בסגירת מבנה נתונים זה יש לסגורhandles לתהליך ולחוט, שנמצאים בתוך מבנה הנתונים.

חוטים: Threads:

חוטים באותו תהליך משתפים קוד , נתונים ומשאבים, אך לכל אחד רגיסטרים ומחסנית משלו . תהליך בברירת המחדל יהיה עם חוט אחד , אך יכול להיות multi-threaded, מעין תתי-משימות הרצות "במקביל". חוט יכול להיות במצב התחלתי, מוכן לריצה במעבד, רץ במעבד, ממתין (למשל לאירוע כלשהו שיתרחש), או סיים את פעולתו. TerminateThread: סיום חוט כמו סיום תהליך (מחזיר קוד יציאה), ניתן גם עליו להשתמש ב-WaitFor...

Process Control Block: PCB:

המערכת מחזיקה מבנה נתונים לתהליך , הכולל מזהה, רשימת handles לקבצים פתוחים, רשימת Thread control block – TCB, שהיא רשימה משורשרת של TCBים לחוטים השונים, הכוללים למשל PC, רגיסטרים וכו'. כאשר חוט נמצא במצב running ומופסקת פעולתו, ה-PC והרגיסטרים נשמרים ב-TCB המתאים שלו, והוא עובר למצב waiting/ready. חוט אחר מועלה לתוך המעבד (PC, רגיסטרים) ורץ. אם ה-context switching נעשה בין שני חוטים בתהליכים שונים, יש צורך לשנות גם את מפת הזיכרון במעבד (ה-MMU, memory management unit, מתוכנתת מחדש).

<u>סנכרון תהליכים</u>:

אובייקטים שונים יכולים להיות במצב <u>signaled</u>, למשל תהליך. תהליך יכול לקרוא לקריאת מערכת שממתינה עד שאובייקט נהיה signaled (או timeout). כאשר אובייקט מאותת, **<u>כל מי שממתין לו</u>** מקבל הודעה על כך. רשימת אובייקטים ומתי הם במצב signaled:

- <u>תהליך</u>: יהיה signaled כאשר הוא מסתיים. כך, אם רוצים לחכות עד שתהליך מסתיים , קוראים ל-WaitForSingleObject(hProcess...), וממתינים להחזרת תשובה.
- <u>אירוע</u>: יצירת אירוע ע"י CreateEvent(NAME...) (מקבל עוד פרמטרים, כמו מצב התחלתי) מחזיר handle – אם קיים כבר אירוע בשם הזה , יפתח אותו (handle חדש לאותו אירוע, כמו קובץ קיים). ניתן לחכות לאירוע ע"י WaitForSingleObject על ה-handle של האירוע, וניתן לאותת לאירוע ע"י SetEvent(hEvent). כל תהליך שממתין לאירוע ידע על כך . למשל: לאותת לתהליך שממתין לתהליך אחר היוצר בשבילו קלט , שהקלט מוכן עבורו.

אובייקטים לסנכרון:

- <u>Mutex</u>: CreateMutex(NAME...) (מקבל עוד פרמטרים), אם קיים כבר בשם זה , מחזיר handle לאובייקט הקיים (כמו אירוע). כאשר ה-mutex נהיה signaled הוא יכול להתפס ע"י תהליך כלשהו שמחכה לו עם WaitFor... רק תהליך אחד בכל זמן נתון יכול לתפוס אותו , ואז ה-mutex הופך ל-unsignaled. שימוש ב-ReleaseMutex כדי לשחרר אותו – להפוך אותו ל-signaled שוב. שימוש: שמירה על משאב משותף. אם חוט מחזיק ב-mutex, קריאה ל-Wait.. לא חוסמת אותו.
- <u>Semaphore</u>: כמו mutex עם מונה. יצירה: CreateSemaphore(NAME, Counter, MaxValue...). הסמפור במצב signaled, וזמין לתפיסה, כשהמונה שלו גדול מ-0. כך כמה תהליכים יכולים לתפוס אותו עד שיחסם. ReleaseSemaphore(hSem, d..) – מעלה את המונה ב-d.
- CriticalSection: כמו mutex ל-threads באותו תהליך. פקודות: InitializeCriticalSection, ...Enter...,Leave...,Delete. **פירוט למטה (אחרי הפירוט על DLL)**.

פקודת WaitForMultipleObjects: כמו WaitForSingleObject, רק מאפשרת לחכות לכמה handles שיעברו למצב signaled. פרמטרים: מספר ה-handles, מערך של handles, האם לחכות לכולם או עד הראשון מביניהם (TRUE – לחכות לכולם), TIMEOUT.

<u>תרגיל 2: Party</u>:

- Party.exe: יוצר 10 מבנים של process information ל-10 תהליכים של person.exe, וסמפור עם ערך התחלתי 2 (ומקסימום 2) ל-showtext.exe. כדי לעצור את התהליכים: בקבלת anykey מוגדר אירוע ומופעל, ואותו אירוע יהיה הטריגר בכל תהליכי person.exe לסיומם. אז פשוט ממתינים לסיומם עם WaitForMultipleObjects (ולבדוק שמחזיר WAIT_OBJECT_0, שלא יצא מקוד שגיאה אחר כלשהו). לאחר מכן סוגרים את כל ה-handles.
- Person.exe: כל תהליך כזה שמופעל יוצר (או פותח) את אירוע הסגירה (שמופעל ב-party.exe), ואז מתחיל: יוצר (או פותח) את הסמפור ל-showtext.exe, קורא עליו ל- WaitForSingleObject (אם תפוס לגמרי , ימתין עד שמקום אחד ישתחרר). כשמקבל אישור יוצר תהליך ל-showtext.exe, מחכה לתהליך שיסתיים (to be signaled) עם Wait..., משחרר את הסמפור של showtext (וסוגר את ה-handle שלו), ואז חוזר לבדוק האם אירוע הסגירה קרה. אם לא, ממשיך בלולאה.
- Showtext.exe: מקבל בפרמטרים שאיתם קורא לו person את ה-ID של ה-person שקרא לו, מגריל מספר מ-5 עד 10 וכותב ב-delay של שניה שורת "person i is talking". כשמסיים, יאפשר ל-person שקר לו להמשיך.

<u>זיכרון וירטואלי</u>:

מערכת ההפעלה מספקת זכרון וירטואלי לתהליכים . המערכת ממפה דפי זיכרון וירטואלי לדפי (מסגרות) זיכרון פיסי (בזיכרון או בטבלת הדפים – הבאה מהדיסק). בזמן נתון רק דפים שהתהליך משתמש בהם נמצאים בזיכרון הפיסי , ודפים משותפים לשני תהליכים ממופים לאותה מסגרת פיסית (כמו קוד של המערכת למשל). דפים לא בשימוש כרגע יאוחסנו בדיסק . הרשאות תהליך בגישה לזיכרון : ביצוע בלבד (ללא שינוי) של קוד, קריאה של נתונים, קריאה/כתיבה בלי ביצוע לנתונים ב-stack/heap. תהליכים יכולים לחלוק זיכרון נתונים ביניהם

שימוש נכון למתכנתים: לא להשתמש ביותר מדי זיכרון וירטואלי, לשים נתונים שמשתמשים בהם יחד במקום קרוב בזיכרון (למשל מערך רציף), לשים קוד ונתונים משותפים ב-DLL כדי לחסוך שכפול.

סיכום system-calls חשובות:

System call: return value	Parameter	Usage
VirtualAlloc:LPVOID	• __in_opt LPVOID lpAddress, - כתובת ההתחלה של המקום שרוצים • __in SIZE_T dwSize, - גודל בבתים • __in DWORD flAllocationType, - MEM_COMMIT, MEM_RESERVE – האחרון רק שומר מקום, לא מאתחל אותו • __in DWORD flProtect – ההגנה על מקום זה, למשל: PAGE_READWRITE, PAGE_EXECUTE	אלוקציה של זיכרון וירטואלי במרחב הזכרון הוירטואלי של התהליך הקורא לפונקציה.
VirtualFree:BOOL	• __in LPVOID *lpAddress*, - כתובת התחלתית ממנה רוצים לשחרר • __in SIZE_T *dwSize*, - גודל לשחרר, בבתים • __in DWORD *dwFreeType* - סוג שחרור: DECOMMIT/RELEASE	שחרור מקום/התחייבות למקום בזיכרון הוירטואלי של התהליך הקורא לפונקציה.

- GetProcessWorkingSetSize : מקבלת handle לתהליך, מצביע למינימום ומצביע למקסימום, ומחזירה בתוכם את ה-working-set המינימלי והמקסימלי של התהליך. תזכורת: working set היא הערכה של מערכת ההפעלה של מס' הדפים בפועל שהתהליך צריך בזכרון הפיסי.
- VirtualLock / VirtualUnlock : מקבלת כתובת וגודל בבתים, ונועלת אזור זה בזכרון כך שלא יפונה לדיסק עד שיתבצע עליו שחרור.

<u>Memory Mapped Files</u>: <u>שיתוף זכרון לתהליכים</u>:

הקובץ Pagefile.sys הוא קובץ שחלקיו ממופים לזכרון בעת צורך. ניתן למפות קובץ כלשהו לזכרון ולגשת אליו מתהליכים. דרך נוספת היא להשתמש ב-pagefile.sys ודרכו למפות זכרון משותף , כאשר לקובץ זה אין גישה מהדיסק , גישה אך ורק לחלק ממנו שממו פה ע"י המערכת לזכרון (דרך פחות נוחה באופן כללי).

סיכום system-calls חשובות:

System call: return value	Parameter	Usage
CreateFileMapping:HANDLE	• __in HANDLE hFile, - INVALID_HANDLE_VALUE • __in_opt LPSECURITY_ATTRIBUTES lpAttributes, - NULL • __in DWORD flProtect, - PAGE_READWRITE – מאפשר כתיבה וקריאה • __in DWORD dwMaximumSizeHigh, - החלק העליון של גודל הקובץ שרוצים למפות. אם זה קטן,אז כנראה שיהיה 0 • __in DWORD dwMaximumSizeLow, - החלק התחתון, אם הקובץ קטן אז כאן זה המקום לתת לו ערך מספרי כלשהו של הגודל הרצוי, כמו גודל מקסימלי של תור בדוגמאות שניתנו בכיתה ובשיעורי הבית. • __in_opt LPCTSTR lpName – שם הקובץ	יוצר קובץ בזכרון הפיסי ומחזיר handle <u>הערה</u>: אם כבר קיים file mapping עם אותו שם, לא ייצור חדש. השם משותף לכל מרחב השמות, כמו mutex או event וכו'.
MapViewOfFile:LPVOID	• __in HANDLE hFileMappingObject, - הקובץ אותו ממפים • __in DWORD dwDesiredAccess, - למשל: FILE_MAP_WRITE אילו הרשאות יהיו לתהליך באובייקט זה • __in DWORD dwFileOffsetHigh, • __in DWORD dwFileOffsetLow, שני הפרמטרים הנ"ל קובעים את ה-offset מתחילת ה-file mapping שבו יתחיל ה-view • __in SIZE_T dwNumberOfBytesToMap – לרוב , גודל הבתים שרוצים למפות נבחר את גודל האובייקט כולו, כמו בדוגמאות בכיתה	מחזיר מצביע ל-file mapping שמקבל כפרמטר עבור התהליך שקורא לפונקציה. לוקחים את ערך הפונקציה כ- *BYTE, ונניח כמו בשיעור וזהו מבנה נתונים מסוג QUEUE, יוצרים מבנה נתונים חדש שהוא casting של ה-*BYTE שקיבלנו לסוג זה של מבנה נתונים, וכך ניתן לעבוד על הזיכרון המשותף.
UnmapViewOfFile:BOOL	• __in LPCVOID lpBaseAddress – הכתובת שמיפינו קודם	שחרור המיפוי שביצענו ע"י MapViewOfFile

<u>תרגיל 3 : Shared-Queue, Reader & Writer</u> :

בכל אחת מהתוכניות reader/writer : מופיעות כל הגדרות ה-QUEUE. כמו המתואר לעיל בהתחלה יש יצירה של תור משותף בזיכרון ע "י קריאה ל-CreateFileMapping ולאחריה קריאה ל-MapViewOfFile כדי להצביע אליו מהתהליך הנוכחי . מחזיקים שני סמפורים – לקורא ולכותב, ובתחילת הריצה יוצרים את שניהם:

- ReadSem : מונה נוכחי בגודל מס' האיברים הנוכחי בתור, מקסימום – גודל התור המקסימלי. יוכל לקרוא עד שלא יהיו איברים לקרוא
- WriteSem : מוני נוכחי בגודל מס' האיברים <u>הפנויים</u> כרגע בתור, מקסימום – גודל התור המקסימלי. יוכל לכתוב עד שלא יהיה מקום בתור.

ה-writer : מחכה לסמפור שלו, וכשמקבל: מקבל קלט, כותב אותו לתור ו<u>משחרר 1 בסמפור של הקורא</u>.

ה-reader : מחכה לסמפור שלו, וכשמקבל: קורא מהתור, <u>משחרר 1 בסמפור של הכותב</u> וכותב למסך עם המתנה.

הערה לגבי מימוש processQueueCommand: פונקציה זו אחראית על ביצוע פעולות על התור. התור מחזיק mutex, ולפני כל ביצוע פעולה של שינוי התור, פונקציה זו קוראת ל-Wait... כדי לתפוס את ה-mutex לפני שינוי התור.

לבסוף משחררים את כל מה שצריך: ה-mutex של התור, unmapViewOfFile, שחרור ה-file mapping ושחרור שני הסמפורים.

DLL's:

כמו שמשתמשים ב-file mapping לנתונים משותפים, כך ניתן להשתמש ב-DLL (Dynamic Linked Libraries) לקוד משותף בין תהליכים. קובץ DLL מכיל רשימת פונקציות מקומפלות ומוכנות לשימוש, וטבלת ייצוא: שם/מספר הפונקציה עם מיקומה בקובץ ה-DLL. יתרונות השימוש: שומר זכרון – מועלה רק פעם אחת לזיכרון, reuse לקוד, הקטנת חבילת התקנה ועדכון קל – החלפת הפונקציה ב-DLL בו מופיעה בלבד.

שתי שיטות:

- Run-time binding: החלטה בזמן ריצה איזו ספריה ואילו פונקציות מתוכה לשימוש, גמיש אך יותר קוד.
- Compile-time: החלטה בזמן קומפילציה על הפונ' והספריה, פחות גמיש אך פחות קוד.

קובץ ה-DLL: יכיל בלוק {} "extern "C שבתוכו יכיל הצהרות לייצוא פונקציות באופן הבא: <method_sign> BOOL __declspec(dll**export**). יכיל גם DLL-main: BOOL APIENTRY DllMain(HMODULE hModule,DWORD ul_reason_for_call,LPVOID lpReserved). בשאר גוף הקובץ יכיל את הגדרות הפונקציות לייצוא.

Compile-time: בשיטה זו כל שצריך הוא שבתוכנית יהיה בלוק {} "extern "C ובתוכו <method_sign> BOOL __declspec(dll**import**) לפונקציות שרוצה לייבא מה-DLL. השימוש בפונ' אלה בשאר התוכנית כרגיל, כאילו הוגדרו בתוכנית עצמה.

Run-time: תוכנית שמשתמשת בשיטה זו: קריאה ל-loadLibrary כדי לטעון את הספריה. ניתן להשתמש בתחילת התוכנית ב-typedef לפונקציה, אליה נעשה casting כאשר נייבא את הפונקציה הנדרשת מהספריה. ע"י GetProcAddress(hModule,"<func_name>") מייבאים את הפונקציה עצמה, ואז ניתן להשתמש בה. לבסוף משחררים את הספריה ע"י קריאה ל-FreeLibrary(hModule).

System call: return value	Parameter	Usage
LoadLibrary:HMODULE	• _in LPCTSTR lpFileName: שם הספריה ממנה מייבאים	קריאה לספריית DLL באופן דינמי כדי להוציא ממנה פונקציה ב-run-time. מחזיר handle למודול של ספרית ה-DLL הרלוונטית.
GetProcAddress: FARPROC	• _in HMODULE hModule, - מזהה לספריה הרלוונטית • _in LPCSTR lpProcName – שם הפונקציה שרוצים לייבא ממנה או מספר הפונקציה (בתרגול ראינו דוגמא עם שם פונקציה)	

Threads:

לחוטים באותו תהליך יש אותו מרחב כתובות: חולקים code, heap, אך לכל אחד PC ו-stack משלו. הפעלה לסירוגין ע"י מערכת ההפעלה. לתהליך תמיד יש חוט ראשי שמקושר אליו, זה שמוחזר לו handle ב-CreateProcess. ביצירה של חוט אחד הפרמטרים הוא פונקציה, שהיא תרוץ כחוט נפרד. כניסה למשאבים משותפים בתהליך ע"י מספר חוטים צריכה להיות מסונכרנת. כמו תהליך, Thread יהיה signaled כאשר הוא מסיים את ריצתו. ניתן לקרוא למשאבים משותפים כמו אירועים אוmutex בשם NULL.

CriticalSection: לסנכרון בין חוטים של אותו תהליך בלבד. למשל: הגדרת CRITICAL_SECTION guard. לאחר מכן הגדרת פונקציה שתרוץ בחוטים שונים, הבנויה כך:

- בתחילת הפעולה: **EnterCriticalSection**(&guard) – תפיסת המשאב, או המתנה עד שיתפנה אם תפוס.
- בסוף הפעולה: **LeaveCriticalSection**(&guard) – שחרור המשאב.

קריאה ב-main של התהליך: בתחילת הפעולה יש לקרוא ל-**InitializeCriticalSection**(&guard) לפני יצירת החוטים. לאחר מכן יצירת החוטים: CreateThread(NULL,0,<func_name>,&<list_of_params>,0,NULL) – מה שחשוב כתוב מפורשות. אם משתמשים באירוע כדי לעצור את כל החוטים, נשתמש ב-WaitForMultipleObjects על רשימת ה-handles של החוטים, עד שכולם יסתיימו, ורק אז נסיים את התהליך, תוך סגירת כל ה-handles ו-**DeleteCriticalSection**(&guard).

תרגיל 4: שדרוג תוכניות reader, writer מתרגילים קודמים:

- שימוש ב-DLL ליצירת התור ומחיקת התור (כל החלק עם ה-CreateFileMapping וכו'). הכנסת ה-mutex המגן על שינוי התור ושני הסמפורים לקריאה ולכתיבה לתוך מבנה הנתונים של התור.
- שימוש בשני DLL ל-lifo ול-fifo, שמייבאים ומייצאים את הפונ' הנ"ל, וכל אחת מממשת בעצמה ומייצאת את push ו-pop בהתאם ללוגיקה.

- Progress: ספרית DLL עם פונ' פנימית שמהווה פונ' thread. מייצאת שתי פונ': StartProgress – יוצרת אירוע לעצירת ה-progress-bar ומפעילה חוט של כתיבת ה-progress-bar, StopProgress – מאותת לאירוע וגורמת לעצירת החוט, וממתינה: Wait... עד שהחוט יסיים פעולתו לפני יציאה.
- Reader/Writer: ייבוא ב-compile-time את הפונקציות של progress, וב-run-time את הפונקציות של ה-QUEUE, בהתאם למה שמקבלים בקלט (lifo או fifo – לכן הצורך בייבוא דינמי). פעולתם דומה למה שהיה בתרגיל הקודם, רק שלפני ביצוע בפעולה קוראות ל-StartProgress ולאחר ביצועה ל-Stop. במימוש הפעולות עצמו (ב-DLLים) יש מימוש מוגן עם סמפורים ו-Mutex לשינוי התור.

Networking:

פעולות חשובות:

פעולה/מושג	הסבר
DHCP	Dynamic host configuration protocol – כשמחברים מחשב לרשת, הוא צריך לקבל כתובת IP, default gateway ועוד פרמטרים. שרת ה-DHCP, router מקומי למשל, מקבל תשדורת מהחיבור החדש ומקצה כתובת IP וכתובות אחרות לכל השירותים הנדרשים (DNS, default gateway וכו'). כדי לראות את שרת ה-DHCP: ipconfig /all.
DNS	Dynamic name system – לכתובות IP יש שמות סימבוליים, כמו www.tau.ac.il. שרתי DNS אחראים על תרגום השמות לכתובות IP. כתובת שרת DNS מתקבלת ע"י DHCP, וטבלאות תרגום DNS מוטמנות בכל מחשב.
ping	פקודה ששולחת לכתובת ה-IP המצויינת packets בגודל 32 בתים כדי לבדוק תגובה ומחזיר תרגום DNS מעודכן.
tracert	בהינתן פרמטר כתובת IP/שם, מחזיר את מסלול הניתוב לכתובת זו (כל הצמתים במסלול).
route	route print – מדפיס את טבלת הניתוב המקומית.
arp	Address resolution protocol – ה-router מחליט מה הצומת הבא במסלול שליחת חבילה ליעד כלשהו לפי טבלת הניתוב שלו. הוא יודע כתובת IP, אך בשביל האתרנט זקוק לכתובת פיסית. קריאה כזו משדרת את ה-IP בפרוטוקול ARP, ומקבלת חזרה ממי שה-IP שייך לו את הכתובת הפיסית הנדרשת.
nslookup	מחזירה תרגום של שם סימבולי לכתובת IP (משרת ה-DNS)
netstat	פותח פורטים של UDP/TCP. הרצה של netstat תתן את רשימת הפורטים מסוג UDP או TCP: כתובת+פורט מקומי, כתובת+פורט זר ומצב (למשל established).

כיצד עובד הדפדפן:

- מחברים את המחשב ל-LAN, למשל לאתרנט.
- מקבלים דרך פרוטוקול DHCP כתובת IP, gateway וכתובת DNS.
- מכניסים URL לדפדפן. ה-http בהתחלה מציין את פרוטוקול התקשורת בשכבת האפליקציה.
- חלק מהכתובת (החלק של ה-host, למשל www.tau.ac.il) מתורגם ע"י שרת ה-DNS לכתובת IP.
- טבלת הניתוב אומרת לאיזה gateway לנתב את החבילה.
- שימוש ב-ARP כדי לגלות מהי הכתובת הפיסית (MAC address) של אותו IP שנבחר.
- יצירת בקשת HTTP: GET default.html HTTP/1.1.
- התחלת session של TCP, החיבור הוא בפורט 80. לחכות ל-ACK (וידוא הגעה) ושליחה מחדש אם צריך.
- קבלת תגובה מהשרת – העלאת הנתונים המתקבלים על המסך ובקשת אובייקטים נוספים כמו תמונות. לאחר מכן ניתוק.

Windows Sockets:

שימוש ב-sockets לתקשורת בין תהליכים, כאשר WinSock הוא ה-API לכך. מאפשר העברה ברשת דרך פרוטוקולי שכבת העברה, הנפוצים ביותר הם TCP ו-UDP. ב-TCP: כל אחד מהמחשבים מזדהה ע"י כתובת IP, כל אפליקציה ע"י פורט - והחיבור הוא ע"י שני sockets מחוברים (אחד לכל אפליקציה). ה-sockets מתחלקים לשני תפקידים:

- שקעים ליצירת קשר.
- שקעים להעברת מידע.

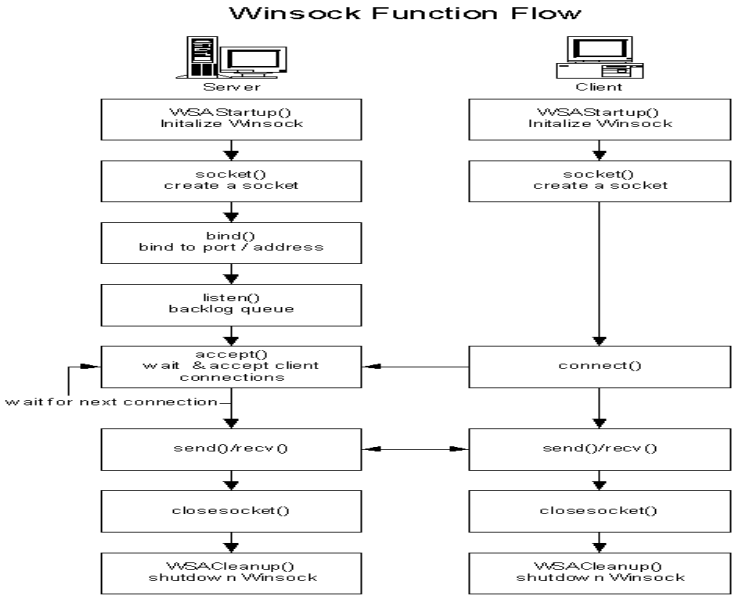
דוגמת שרת/לקוח: **שרת**:

- בתחילת הפעולה צריך לאתחל את מנגנון ה-windows sockets ע"י WSAstartup.
- יצירת listen socket: SOCKET hListenSoc = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP).
- הכנת הכתובת לחיבור לשקע: משתנה מסוג sockaddr_in בעל פרמטרים: sin_family = AF_INET, כתובת IP ופורט.
- חיבור ה-listen socket ע"י קריאה לפקודת bind על ה-listen socket והכתובת שיצרנו.

- התחלת הקשבה: listen(hListenSoc,1)
- לולאה על המתנה לקבל קשר מלקוח ויצירת תקשורת עמו: יוצרים מבנה נתונים מסוגsockaddr שיחזיק את הנתונים על הלקוח, ושקע ללקוח ע"י accept: SOCKET hRecvSoc=accept(...) עם הפרמטרים: שקע ה-listen של השרת ומבנה הנתונים שהרגע בנינו לכתובת הלקוח.
- מכינים חוצץ לנתונים שמתקבלים. כאשר יש חיבור ללקוח, משתמשים בפקודת recv המקבלת מהשקע hRecvSoc בתים עד סיום.
- בסיום סוגרים את השקעים ו-WSACleanup.

לקוח:

באופן דומה, מאתחל את מנגנון ה- sockets ומכין כתובת לשרת. לאחר מכן יוצר socket וקורא ל-connect על השקע עם כתובת השרת, כדי להתחבר לשרת דרך שקע זה. לאחר מכן מכין מבנה נתונים בבתים לתוכן שישלח לשרת, ושולח ע"י קריאה לפקודת send עם השקע והתוכן. בסיום סוגר את השקע ומריץ WSACleanup.



סיכום:

- Bind: קושר שקע עם כתובת מקומית.
- Listen: שם שקע במצב האזנה לקשר מתקבל.
- Accept: מתיר קשר לתקשורת המתקבלת בשקע.
- Connect: יוצר תקשורת לשקע מסוים לפי כתובתו.
- Send: שולח מידע דרך שקע מחובר.
- Receive: מקבל מידע משקע מחובר.

תרגיל 5: Server/Client: אותו רעיון של הדוגמא המתוארת לעיל.

Windows Security:

אבטחה ברמת אובייקט: בדיקת המשתמש וסוג הגישה המבוקשת מול הרשאות אובייקטים, כגון: קבצים, תהליכים/חוטים, אובייקטי סנכרון, דפי זכרון. למשל, CreateFile מבקשת הרשאות כגון: FILE_ALL_ACCESS, FILE_READ_DATA. לתהליכים/חוטים יש מטבע אבטחה שקשור אליהם: מבנה נתונים המכיל (דברים עיקריים):

- SID: מספר מזהה של משתמש/קבוצה.
- רשימת הרשאות מערכת.
- ערכי ברירת מחדל (כשיוצרים אובייקטים עם NULL, כלומר בלי פרמטרי אבטחה מסוימים).

ACL: Access control list:

הדרך של windows לתאר הרשאות לאובייקטים: רשימה של אובייקטים מסוג ACE (Access control entrie), שלכל אחד הערכים:

- סוג: Allow/deny.
- SID של משתמש או קבוצה.
- Access rights mask – ספציפי לאובייקט שה-ACL שייך לו. למשל: כתיבה+קריאה.

System call: return value	Parameter	Usage
OpenProcessToken:BOOL	• __in HANDLE ProcessHandle, - מזהה לתהליך שעבורו פותחים את המטבע • __in DWORD DesiredAccess, - מסיכת הגישה הרצויה למטבע • __out PHANDLE TokenHandle – יצביע למזהה למטבע	פותח את ה-Access token עבור התהליך המבוקש.
GetTokenInformation:BOOL	• __in HANDLE TokenHandle, - מזהה למטבע • __in TOKEN_INFORMATION_CLASS TokenInformationClass, - קבוצת המידע שרוצים להשיג, למשל: TokenUser: user accountof the token • __out_opt LPVOID TokenInformation, - מצביע לחוצץ שיכיל את המידע שהפונקציה תחזיר, על סוג המידע המבוקש בפרמטר הקודם. • __in DWORD TokenInformationLength, - גודל הפרמטר הקודם • __out PDWORD ReturnLength – לא חשוב	מחזירה סוג מסוים של מידע מ- access token. למשל, מידע על TokenUser: מחזיק מידע על ה-SID של ה-user.

LookupAccountSid: הפרמטרים החשובים שמקבל הם SID של משתמש/קבוצה (למשל שדה ה- SID של TokenUser), ומוציא (לתוך פרמטרים שמקבל) את שם המשתמש וה-domain שבוא נמצא.

דוגמא ליצירת אירוע: יוצרים אירוע עם NULL כפרמטר אבטחה. ה-DACL (discretionary access control list) של החוט הקורא מועתק לאובייקט האירוע. יצירת האירוע שוב מתהליך אחר מבקשת גישה , ואז ה-ACL של האירוע מושווה עם ה- SID ב-thread token שקרא לו, ונעשה חיפוש לחוק ACE – allow/deny עם אותו SID לפעולה המבוקשת על אובייקט האירוע

Structured Exception Handling:

Exception: אירוע שקורה בזמן ריצת תוכנית, ודורש ביצוע קוד מחוץ לזרימה הנורמלית של התוכנית. סוג הטיפול בחריגים ב-windows הוא frame-based, כלומר טיפול בחריג בקטע קוד ספציפי. בלוק try-except:

__try { guarded body of code } __except (filter expression) { execution-handler block }

הפילטר בנוי כך שאם תופס קוד שגיאה מסוג מסוים , נבחר שיריץ את: EXCEPTION_EXECUTE_HANDLER, כלומר יריץ את הבלוק הבא , ואחריו ימשיך בזרימת התוכנית . האופציה השניה היא: EXCEPTION_CONTINUE_SEARCH – המשך חיפוש בסקופ הבא של התוכנית . למשל: בפונקציה יהיה קטע קוד עטוף עם פילטר:

__except((GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION)?EXCEPTION_EXECUTE_HANDLER:EXCEPTION_CONTINUE_SEARCH)

במקרה זה אם קוד השגיאה בקטע העטוף היה EXCEPTION_ACCESS_VIOLATION (חריגה לאזור אסור בזיכרון , יכול להיות גם חלוקה באפס למשל), אז טפל בו כאן. אחרת, המשך לסקופ הבא. קריאת הפונקציה תהיה עטופה ב-try-except ב-main, והוא יוגדר תמיד בכל מקרה לטפל באירוע (פילטר ה-except יהיה EXCEPTION_EXECUTE_HANDLER), והוא יטפל בחריג.

סיכום פילטרים:

- EXCEPTION_CONTINUE_EXECUTION: המשך את ריצת התוכנית מהנקודה בה קרה החריג.
- EXCEPTION_CONTINUE_SEARCH: החריג לא מטופל, המשך חיפוש טיפול במעלה הסקופ (המחסנית).
- EXCEPTION_EXECUTE_HANDLER: החריג זוהה, טפל בו ע"י הרצת בלוק ה-Except__ והמשך את הריצה שאחריו.

תרגיל 6: exception handling: אותו רעיון כמתואר לעיל, נעשה שימוש גם בחריג EXCEPTION_INT_DIVIDE_BY_ZERO.