Windows Subsystem for Linux - המכונה שתמיד רציתם

מאת יהונתן אלקבס

הקדמה

Windows 10 Hyper-V מאפשר לנו לעשות דברים די מיוחדים. החל מלהריץ VMs שונים, לשתף מכשירים (כוננים, מדפסות, רמקולים ועוד) באמצעות RDP, ליצור virtual switch על מנת לחבר את המכונה לאינטרנט ולהשתמש ב-Windows Shell בתור ה-UI בעוד למעשה הקרנל יהיה של Linux distors ביחד עם ה-CMD; הכל באותה מכונה תחת אותה חומרה, צד אל צד והכי חשוב - ב-native speed.

אז מה נעשה היום?

התשובה הקצרה - נכיר את WSL ונריץ מספר subsystems בו זמנית על גבי מכונת ה-Windows שלנו.

התשובה הארוכה - נבנה ונבצע קומפילציה לקרנל לינוקסי בהתאם לדרישות ספציפיות שנגדיר על גבי מכונת Ubuntu. לאחר מכן נעבור לעולם ה-Windows ונריץ את הקרנל שבנינו על תשתית WSL2 ולאחר שנוודא כי הליבה שבנינו אכן עובדת נעבור להסתכל על היתרונות והחסרונות של כל הסיפור והאם זה באמת אפשרי להריץ מספר subsystems במקביל מבלי לפגוע בביצועים כמו שצוות ה-marketing של Windows אוהב לפרסם + איך זה באמת נראה להריץ מספר subsystems מבחינת WSL1 לעומת WSL2?

בקיצור, מטרת המאמר היא להסביר בצורה פרקטית (וקצרה!) על Windows hypervisor תוך כדי בניה והרצה של windows shell UI על CLI של ליבה והפצה של לינוקס; כל זאת בצמוד לידע התיאורטי הנדרש בשביל להבין את מה שקורה. עם סיום המאמר אולי אפילו אצליח לשכנע את חלקכם ש-WSL הוא פתרון מעולה וכמעט לכולם יש מה להרוויח מהוספתו לארגז הכלים. חשוב כבר עכשיו לומר כי המסמך מניח שלקורא ידע מקדים במערכות הפעלה.

למה לי לעשות דבר כזה?

למה שנרצה להנדס קרנל לינוקסי משלנו? מה רע באלפי הגרסאות שיש היום? לינוקס היא (הוא?) בסה"כ רעיון - מערכת הפעלה חינמית, פתוחה לשימוש ולתרומה בקוד פתוח. המבנה המודולרי שלה גורם לה להיות גמישה וניתנת לשינוי. כל חלק מרכיבי המערכת מחולק למספר packages שניתן בקלות להוסיף, למחוק או להחליף במשהו אחר; שום חלק במערכת ההפעלה לא יוצא דופן - **גם לא הקרנל עצמו.**

אוקיי, אז למה שמישהו ירצה לקמפל ולהריץ קרנל של לינוקס?

המקרה הראשון הוא כאשר צריכים קרנל חדש יותר מאשר יש ל-distro שנמצא בשימוש כרגע (במיוחד כשמדובר בהפצות יציבות כמו Debian stable, CentOS) בגלל שהקרנל שלו לא תומכת בדרייבר ו\או פיצ'ר שאנחנו צריכים בצורה ספציפית. המקרה השני הוא כשאתם מרגישים שיש לכם **bloated kernel** עם מלא רכיבים ופיצ'רים שאתם פשוט לא צריכים - בין אם זה תמיכה במערכת קבצים מסויימת שאתם לא משתמשים בה בכלל או בין אם זה תמיכה ברכיבי חומרה (אין שום סיבה שהקרנל שלכם יתמוך ב-400 סוגי CPUs כשיש רק אחד במחשב שלכם). קרנלים כלליים שמגיעים עם כל הפצה צריכים לתמוך בכמה שיותר חומרה בשביל שבזמן ה-boot הם יוכלו לזהות אילו רכיבים מחוברים למחשב ויטענו את ה-Modules המתאימים. עכשיו, למרות שהורדת תמיכה בחומרה לא תוריד משמעותית את מהירות הקרנל בשוטף היא כן תוריד בצורה ניכרת את זמן ה-boot.

רעיון נוסף להורדת התמיכה בחומרה וממשקים מיותרים הוא העלת רמת האבטחה. ההיגיון פה הוא פשוט, אם יש לך פחות רכיבים במערכת - לתוקף יש פחות משטח תקיפה. אנשים שמפזרים משפטים כמו "תן לזה לרוץ, זה בכל מקרה משתמש בכלום משאבים" הם פלומניקים. בכל בחירה יש הרבה יותר שיקולים מאשר רק משאבים.

סיבה אחרת, וכנול הנראה גם הכי נפוצה, היא **הוספת פיצ'רים** שפשוט אין לנו עם הקרנל הנוכחי. לכל אחד מאיתנו יש את ההרגלים וסביבת העבודה שהוא מרגיש הכי בנוח לעבוד איתה וקינפוג נכון של הקרנל יאפשר לנו להרגיש בבית. במהלך ההכנה למאמר נתקלתי בבלוג של מישהו שהעובדה ש-WSL מאפשר כעת לתמוך במערכת הקבצים ZFS הלהיבה אותו כל כך שהוא היה חייב לכתוב על זה.

מה בנוגע לחקירת מודלים ספציפיים? בין אם אתם חוקרי אבטחה שרוצים לבדוק פיצ'רים ספציפיים של module מסוים בקרנל ואיך הוא משפיע על אלמנטים אחרים במ"ה או בין אם אתם מפתחים שרוצים לודא שסביבת הפרודקשן שלכם נקייה וסטירילית ככול הניתן; משחק עם המודלים האלו חייב להתבצע לפני וזה בדיוק השלב בו ביצוע קומפילצייה לקרנל יהיה האפקטיבי ביותר.

הסיבה שיש אלפי גרסאות ללינוקס רק תומכת בטענות האלו.

סדר רבותיי סדר

Subsystem הוא אוסף של תהליכים וקבצים שניתן בעזרתם להתממשק עם הקרנל ושירותים של מערכת ההפעלה עבור יישומים במערכת. כל executable חייב שיהיה לו subsystem בשביל לרוץ (חוץ ממקרים מיוחדים), זוהי הדרך של מערכת ההפעלה לחשוף חלק משירותי הקרנל עבור תהליכים ב-User mode.

ההתממשקות של תהליכים ב-User mode עם מערכת ההפעלה והחומרה שעליה היא רוצה לבצע את הפעולות לרוב תתבצע דרך ה-subsystem. דוגמה טובה לכך היא Win32 API שבאמצעותו נוכל להתממשק עם מ"ה וה-Native API שמהווה את הקוד שתהליכים ב-user mode יוכלו להשתמש בו עם הרשאות קרנל (באמצעות system calls). שני ה-APIs האלה הם חלק מהמארז שאנחנו קוראים לו **Win32 Subsystem**, ה-subsystem של Windows.

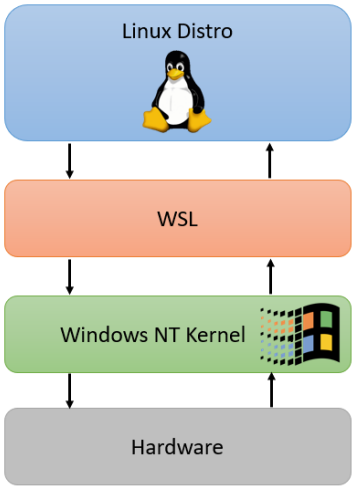
להרחבה מומלץ לפנות למאמר המדהים "The New Processes of Linux" של רז עומריי מגליון 103.

בקיצור, על מערכת ההפעלה של Windows יכולים להיות עוד subsystems - עוד מארזים כאלה שנוכל להשתמש בהם כדי להציג דברים באופנים שונים ולהתממשק עם החומרה ועם המערכת באופן שונה.

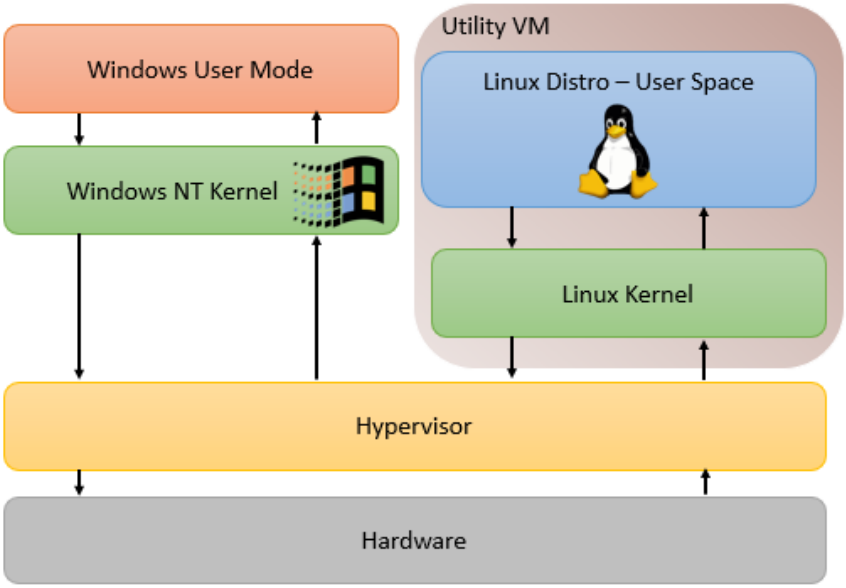
מה? מה??

נתחיל מההתחלה:

הארכיטקטורה של Windows NT kernel תוכננה מראש לתמוך במספר subsystems אשר רצות **במקביל**. במקור היא תוכננה לתמוך ב-subsystems של OS/2 וב-POSIX (שהוחלפו די מהר) וכמובן ב-subsystem של Windows - Win32. בכל מקרה כזה, ה-subsystems תוכננו לרוץ מעל הקרנל של Windows NT. זה אומר שהפצה כלשהי כמו Ubuntu למעשה יושבת מעל Windows subsystem אשר בתורה יושבת מעל הקרנל של Windows NT. השכבה הנוספת הזו של אבסטרקציה ותרגום מוסיפה תווך נוסף לעומת מה שיהיה לנו בדר"כ כשאנחנו מדברים על הרצה של native Linux system במכונה לבד. כתוצאה מכך, כפי שניתן לשער, הביצועים לא היו משהו בכלל - בכל פעם שהתבצעה system call של לינוקס היא הייתה צריכה להתרגם ל-system call של Windows NT.



WSL עשה את העבודה שלשמה כתבו אותו - הרצת אפליקציות ו-distros של לינוקס על מכונת Windows. אבל, וזה אבל גדול, ה-overhead של שכבת התרגום עיכב את כל העיניינים משמעותית וזה פשוט לא היה מספיק בשביל לגרום לאנשים להשתמש ב-feature הזה ש-Windwos דאגו להראות לכולם (אבל היי, זה היה אחלה גימיק שיווקי). בשביל לפתור את הבעיה הנ"ל (ועוד כמה דברים) מייקרוסופט פיתחו את WSL2. בגרסה הזו ה-NT kernel יוצא מהתמונה לגמרי והקרנל של לינוקס מדבר **ישירות** עם הוירטואליזציה של החומרה באמצעות ה-hypervisor.



מלבד הנוכחות של ה-hypervisor עצמו, אפשר למעשה לקבל רמת ביצועים מאוד דומה של ה- Linux subsystem כאילו היינו מריצים את אותה הפצה של לינוקס natively על החומרה עצמה[1]. WSL2 רץ באופן דומה ל-VM אבל מנוהל לגמרי מאחורי הקלעים ומספק 100% תאימות עם system calls בגלל שהוא נותן לנו את ה-Linux kernel עצמו - זה תכל'ס אותו הקוד אחד לאחד. למה שהאפליקציות לינוקס שיש לי יבזבזו זמן על תרגום syscalls ולא ידברו ישירות עם ליבה לינוקסית שמבינה אותן?

בשביל להמחיש את הרעיון בפרקטיקה - אם ב-WSL1 היה לנו קוד שמבצע disk IO, שכבת הלינוקס הייתה מדברת עם הקרנל של Windows NT ועם מערכת הקבצים של NT ואז ה-I\O calls של Linux היו מתורגמים ל-native NT calls ומועברים הלאה.

זאת בניגוד למצב ב-WSL2 שבו אנחנו אשכרה **מריצים natively את מערכת הקבצים EXT64** אבל בתוך hard disk וירטואלי. לינוקס מדברת ישירות עם הדיסק בדיוק כמו שהיא הייתה עושה אם היה מדובר ב-standalone installation. אם השתמשתם בעבר ב-WSL1 ועברתם ל-WSL2 אני בטוח ששמתם לב כמה מהר פתאום פעולות כמו קריאה וכתיבה לדיסק מתבצעות.

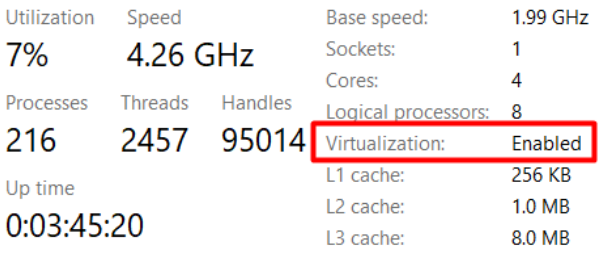
להרחבה על הארכיטקטורה של WSL - גליון 103, רז עומרי "The New Processes of Linux". מומלץ!

בפרק הבא נבנה גרסה משלנו של Linux kernel על Ubuntu ב-WSL2 ואז נחליף את הקרנל ל-Windows ונעבור להריץ את הקרנל שבנינו ב-WSL2 מבלי אפילו לעשות reboot ל-Windows. במילים אחרות, בדיוק כמו שהיינו מבצעים קומפילציה לתוכנית ב-Windows בסביבה Windows-ית אנחנו הולכים להשתמש בלינוקס בשביל לבצע קומפילציה לגרסה custom made של לינוקס ואז להריץ אותה בסביבת WSL.

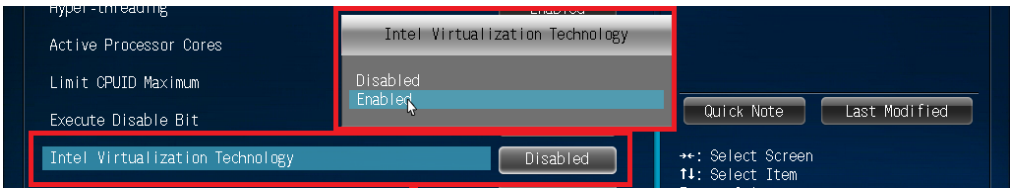
טוב, יאללה לעבודה

הגיע הזמן לבנות קרנל משלנו.

אמרנו ש-WSL2 רץ באופן דומה ל-VM ולכן אני מקווה שזה לא מפתיע אתכם שאנחנו הולכים להסתמך **הרבה** על יכולות הוירטואליצזיה של ה-CPU שלנו (לא משנה אם זה Intel או AMD). לכן, השלב הראשון הוא לודא שהוירטואליזציה מאופשרת - אפשר לבדוק את זה בכמה דרכים אבל הדרך הנוחה ביותר היא באמצעות Task Manager בחלונית Performance:



במידה והשירות לא מאופשר לכם, תצטרכו לבצע reboot, להיכנס ל-BIOS ולאפשר אותו ידנית.



כמו כן, בשביל להריץ את מלוא השירותים של WSL2 צריך לפחות build 1903 של Windows, אישית אני מבצע הכל על גרסה H220 ו-build 19042. אם יש לכם Build מתאים ועדיין אין לכם WSL ניתן להוריד אותו באמצעות הפקודה **wsl -install** או להיעזר בדוקומנטציה.

על מנת להשאיר את המאמר כמה שיותר פשוט מבחינה טכנית אנחנו הולכים לבצע מניפולציות על העתק של קרנל לינוקסי ישר מ-GitHub tree של מייקרוסופט שמאפשר לקרנל לרוץ על Windows hypervisor. באותה מידה היינו יכולים להוריד את הקרנל של Linus Torvals עצמו או על כל קרנל של הפצה אחרת אבל היו לזה השלכות נלוות של הכנת סביבת העבודה לפני ביצוע deployment לקרנל המוגמר שלנו. לא

מדובר במשהו בסובך אבל אני מעדיף לא להיכנס אליו מכיוון שחלק ממטרת המאמר היא להאיר את השימוש ב-WSL.

אני הולך לעבוד עם קרנל בגרסה 4.19.84 של לינוקס אבל בחירה בכל גרסה אחרת שמייקרוסופט אימצה (בשביל לספק תמיכה בסביבת WSL) אפשרי בהחלט.

ראשית נכין את סביבת העבודה שלנו במכונת Ubuntu (נוריד את כלי הקומפילצייה שיתמכו לנו בתהליך ה-build), לאחר מכן, נפנה לגיט של מייקרוסופט בשביל להוריד את קוד המקור של הקרנל עצמה, נוריד קוד נוסף של מייקרוסופט שעוטף את כל הסיפור בצורה נוחה ומגיש את הקרנל ל-hypervisor ובסוף נקנפג ידנית אובייקטים בקרנל.

אני רץ על Ubuntu 20.04 LTS ב-WSL (דיפולטיבי) אבל כל הפצה אחרת ב-WSL אפשרית באותו מאמץ. נתחיל. ראשית נוריד את הכלים שאנחנו צריכים בשביל לבצע קומפילצייה:

```
jon@JoniKamoni:~$ sudo apt install build-essential flex bison libssl-dev libelf-dev
```

בין הכלים שהחבילה מתקינה ניתן למצוא את:

```
The following NEW packages will be installed:
  binutils binutils-common binutils-x86-64-linux-gnu bison build-essential bzip2 cpp cpp-8 dirmngr dpkg-dev fakeroot
  flex g++ g++-8 gcc gcc-8 gnupg gnupg-l10n gnupg-utils gpg gpg-agent gpg-wks-client gpg-wks-server gpgconf gpgsm
  libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl libasan5 libassuan0 libatomic1 libbinutils
  libbison-dev libc-dev-bin libc6-dev libcc1-0 libdpkg-perl libelf-dev libfakeroot libfile-fcntllock-perl libfl-dev
  libfl2 libgcc-8-dev libgdbm-compat4 libgdbm6 libgomp1 libisl19 libitm1 libksba8 libldap-2.4-2 libldap-common
  liblsan0 libmpc3 libmpfr6 libmpx2 libnpth0 libperl5.28 libquadmath0 libreadline7 libsasl2-2 libsasl2-modules
  libsasl2-modules-db libsigsegv2 libsqlite3-0 libssl-dev libstdc++-8-dev libtsan0 libubsan1 linux-libc-dev m4 make
  manpages manpages-dev patch perl perl-modules-5.28 pinentry-curses xz-utils zlib1g-dev
```

אלו ממכם שיצא לפתח בעבר בלינוקס בוודאי מזהים חלק גדול מהכלים. כלים כמו binutils שמהווה GNU assembler ולינקר, g++ ו-gcc שהם קומפיילרים לשפת cpp ו-c, dpkg-dev שמכיל סט כלים נפרד ל-debian, make שנשתמש בו בקרוב ועוד ועוד.

לאחר מכן, נתקין חבילה נוספת של כלים, היא לא הכרחית אלא אם מתכוונים ללכת בכיוון של X/Windows (ממשק גרפי) יותר מאוחר אבל לפי המקורות ה-build הספציפי עובד איתם אז אני לא אחבל במתכון שעובד:

```
jon@JoniKamoni:~$ sudo apt install qt5-default pkg-config
```

נוריד את הקוד של הקרנל מהרפו של מייקרוסופט:

```
jon@JoniKamoni:~$ mkdir Kernel
jon@JoniKamoni:~$ cd kernel
jon@JoniKamoni:~/kernel$ git clone https://github.com/microsoft/wsl2-linux-kernel.git
```

איך שהקרנל עוברת קומפילצייה מושפע בצורה ישירה מקובץ קינפיגורציה שמסופק ל-build process. במקום לנסות לכתוב את ה-config file מאפס אני אערוך את קובץ הקינפוג של מייקרוסופט מכיוון שהוא הכי קרוב למה שאנחנו ככול הנראה רוצים.

בסופו של יום, צריך להבין שכל הקוד כבר נמצא לנו מול העיניים וה-config file הוא מה שמוזן לסקריפט שמבצע build והוא בתורו מושך את ההגדרות של הקרנל בזמן הריצה מקוד המקור של הקרנל.

```
jon@JoniKamoni:~/kernel$ cd wsl2-linux-kernel/
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ ls
COPYING        Kbuild    MAINTAINERS  README              block   drivers   include  kernel  net       security  usr
CREDITS        Kconfig   Makefile     README-Microsoft.WSL2  certs   firmware  init     lib     samples   sound     virt
Documentation  LICENSES  Microsoft    arch                crypto  fs        ipc      mm      scripts   tools
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ cp Microsoft/config-wsl .config
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ vim .config
```

בשביל לערוך את הקונפיגורציה של הקרנל נוכל להשתמש בעורך טקסטואלי כדוגמת vim או (מלח מים מלח מים) nano. הקובץ בנוי מהגדרות ומאפיינים שניתן לשנות בהתאם למה שאנחנו צריכים. כך למשל, אם נרצה לשנות את השם של מערכת ההפעלה שניצור - נערוך את הטקסט שאחראי עליה למשהו שאנחנו יותר מתחברים אליו:

```
#
# Compiler: x86_64-msft-linux-gcc (GCC) 8.2.0
#
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=80200
CONFIG_CLANG_VERSION=0
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_EXTABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
# CONFIG_COMPILE_TEST is not set
CONFIG_LOCALVERSION="-microsoft-standard"
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_BUILD_SALT=""
CONFIG_HAVE_KERNEL_GZIP=y
CONFIG_HAVE_KERNEL_BZIP2=y
CONFIG_HAVE_KERNEL_LZMA=y
CONFIG_HAVE_KERNEL_XZ=y
CONFIG_HAVE_KERNEL_LZO=y
```

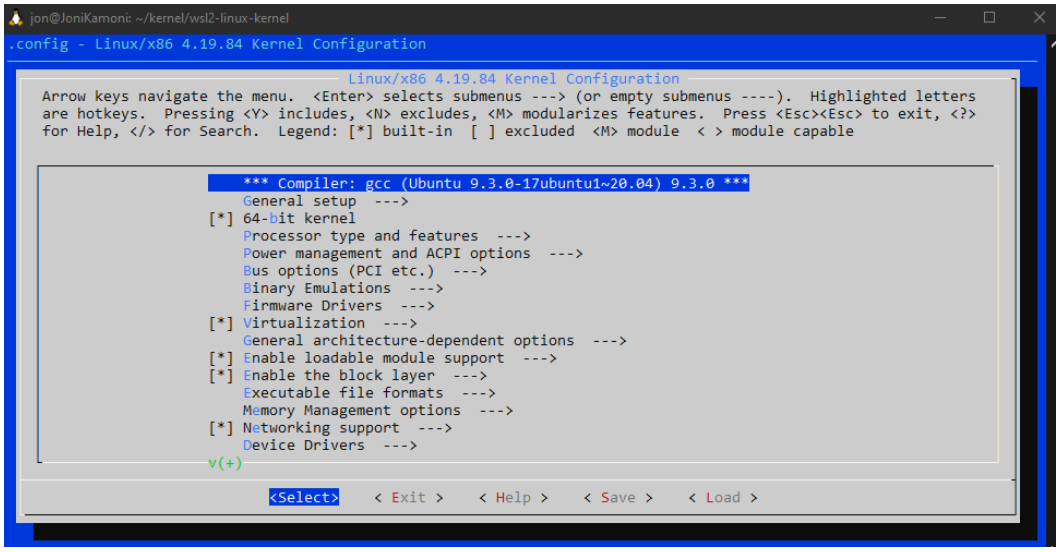
```
#
# Compiler: x86_64-msft-linux-gcc (GCC) 8.2.0
#
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=80200
CONFIG_CLANG_VERSION=0
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_EXTABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
# CONFIG_COMPILE_TEST is not set
CONFIG_LOCALVERSION="-DigitalWhisper-standard"
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_BUILD_SALT=""
CONFIG_HAVE_KERNEL_GZIP=y
CONFIG_HAVE_KERNEL_BZIP2=y
CONFIG_HAVE_KERNEL_LZMA=y
CONFIG_HAVE_KERNEL_XZ=y
CONFIG_HAVE_KERNEL_LZO=y
```

דרך אחרת, נוחה יותר, היא באמצעות ממשק גרפי שיאפשר לנו לעבוד בצורה ויזואלית. בשביל להשתמשבו נצטרך להוריד את הכלי ncurses-dev ולרשום בטרמינל:

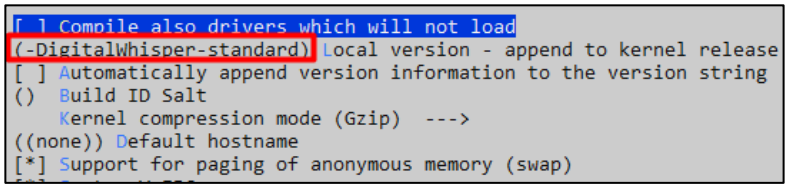
```
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ make menuconfig
```

מה שיפתח:



מכאן, נוכל לבצע כמעט כל שינוי ב-build של הקרנל שנריץ מאוחר יותר.

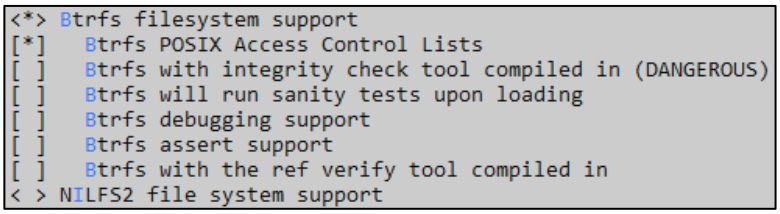
אפשר למשל לראות שהערך של שם מערכת ההפעלה שערכנו מקודם נכתב בצורה תיקנית כי התוכנה הצליחה לפרסר ולרנדר אותו למסך:



איך ממשיכים מכאן?

תלוי מה אתם רוצים לעשות עם הקרנל מאוחר יותר. אם אתם רוצים לחקור בינארים ספציפיים ו\או להבין איך הקרנל מתנהג עם מאפיינים שונים - זה הזמן להוסיף אותם.

← נדגים **הוספה של מודל**: אם למשל רוצים לתת תמיכה למערכת הקבצים **BTRFS** ישירות לקרנל ולאפשר features עבורה אז צריך לנתב ל-"File systems" ומשם לאפשר את הרובריקה של "Btrfs file system support". לכל רובריקה יכולים להיות שלושה מצבים - "**M**", "*****" **וריק**. "M" מסמל שה-feature יבנה כ-module ויטען במקרה הצורך כשמערכת ההפעלה עולה, "*" מסמל שה-feature יבנה **לתוך** הקרנל ותמיד יטען. רובריקה ריקה פשוט לא זוכה להתייחסות מה-build process:



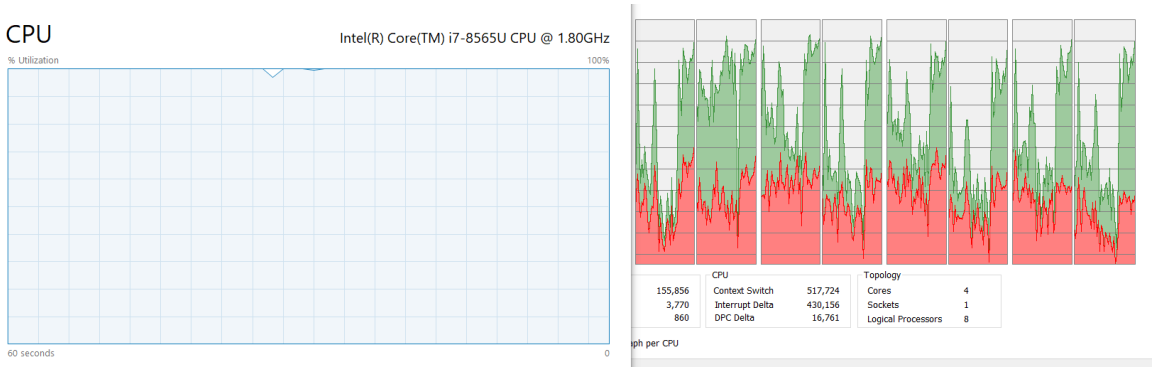
בצורה דומה, אם נרצה שהקרנל שנבנה יתמוך ב-EXT3, EXT4, XFS כי אנחנו רוצים לחקור שם משהו ספציפי ו\או מערכות קבצים נוספות - כל מה שנצטרך זה לאפשר אותן דרך ה-GUI עם features שמתאימים לצרכים שלנו.

הערה: שינוי ההגדרות בחלק מהאופציות יכול לגרום לקריסה של הקרנל בזמן ריצה - זה לא המקום לבדוק דברים בצורה מודולרית. תקראו היטב על כל אופציה לפני שאתם מאפשרים\ מבטלים אותה.

לאחר שנסיים לערוך את קובץ הקינפיגורציה נבצע build באמצעות:

```
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ make -j
```

תהליך הקומפילציה יקח יחסית הרבה זמן, תלוי בחומרה שיש לכם. כשיש הרבה קבצים לקמפל ה-CPU יעבוד קשה ויגיע ל-100% אבל כשהוא יבצע archiving ו-linking (שדורשים רק single core) יהיה אפשר לראות שחלק מהמעבדים יהיו 'מובטלים':



עם סיום הריצה בהצלחה, יווצר קובץ בשם bzImage שמכיל את הקרנל הטרייה שלנו (ביחד עם הדרייברים built in):

```
Kernel: arch/x86/boot/bzImage is ready  (#1)
```

על מנת להשתמש בקרנל החדש שלנו באמצעות WSL אנחנו צריכים להעביר אותו לעולם ה-Windows-י. כאן נכנס היתרון (או חיסרון) של WSL.

WSL מאפשר **שיתוף משאבים** בין ה-subsystems של Windows ולינוקס. ניתן לשתף מערכות קבצים, משתני סביבה, משאבי רשת וכלי CMD (כדוגמת powershell). כלומר, WSL מאפשר לנו לעבוד עם שתי מערכות הקבצים כאילו הן היו אחת. מערכת הקבצים של Windows (NTFS) היא **mounted בתור תיקייה בלינוקס** בעוד שמערכת הקבצים בלינוקס (EXT4 למשל) היא mounted בצורה דומה בסביבת ה-Windows. מאפיין זה יעזור לנו להעביר את הקרנל שבנינו לטריטוריה של Windows - אנחנו נרצה להריץ אותו מ-WSL.

ניצור תיקייה בצד ה-Windows-י ונעביר את הקובץ bzImage מהצד הלינוקסי לשם באמצעות כונן C שהוא ה-mountpoint:

```
PS C:\Users\Jonathan Elkabas> cd /
PS C:\> mkdir wslkernel
```

```
jon@JoniKamoni:~/kernel/wsl2-linux-kernel$ cd arch/x86/boot/
jon@JoniKamoni:~/kernel/wsl2-linux-kernel/arch/x86/boot$ cp bzImage /mnt/c/wslkernel
```

לאחר שנוודא ש-bzImage בתיקייה נוכל לכבות את WSL, לא נצטרך את המכונה שרצה שם עוד. ניצור קובץ בשם wslconfig. שמחזיק קונפיגורציות גלובאליות של WSL ונוסיף אליו את הקרנל שהרגע יצרנו כקובץ WSL:

```
PS C:\wslkernel> wsl --shutdown
PS C:\wslkernel> cd 'C:\users\Jonathan Elkabas\'
PS C:\users\Jonathan Elkabas> notepad .wslconfig
```

כשהקובץ יפתח נכניס את מיקום הקובץ החדש ואת ההגדרות שלו:

```
*.wslconfig - Notepad
File  Edit  Format  View  Help
[wsl2]
kernel=C:\\wslkernel\\bzImage
memory=4GB # Limits VM memory in WSL 2 to 4 GB
processors=2 # Makes the WSL 2 VM use two virtual processors
```

הקובץ מאפשר לנו לקנפג משתנים גלובאליים של WSL ועל ידי כך לשנות את מאפייני ה-VM של WSL2 שאנחנו רוצים רוצים לטעון ל-Hyper-V.

לאחר מכן, כל מה שצריך לעשות זה לצאת מממשק ה-WSL ולהתחבר אליו שוב (אני מבצע הכל באמצעות windows terminal). עם עליית האימג' נוכל לראות שקוד הקרנל שבנינו הוא זה שרץ כעת:

```
jon@JoniKamoni:/mnt/c/Users/Jonathan Elkabas$ cat /proc/version
Linux version 4.19.84-DigitalWhisper-standard+ (jon@JoniKamoni) (gcc version 9.3.0 (Ubuntu
9.3.0-17ubuntu1~20.04)) #1 SMP Tue Mar 16 22:31:45 IST 2021
```

אפשר גם לראות שההגבלות על ה-RAM ועל המעבדים שקבענו בקובץ קונפיגורציה חלים:

```
jon@JoniKamoni:/mnt/c/Users/Jonathan Elkabas$ neofetch
       _,met$$$$$gg.          jon@JoniKamoni
    ,g$$$$$$$$$$$$$$$P.       --------------
  ,g$$P"     """Y$$.".        OS: Debian GNU/Linux 10 (buster) x86_64
 ,$$P'              `$$$.     Kernel: 4.19.84-DigitalWhisper-standard+
',$$P       ,ggs.     `$$b:   Uptime: 4 secs
`d$$'     ,$P"'   .    $$$    Packages: 224 (dpkg)
 $$P      d$'     ,    $$P    Shell: bash 5.0.3
 $$:      $$.   -    ,d$$'    Terminal: /dev/pts/0
 $$;      Y$b._   _,d$P'      CPU: Intel i7-8565U (2) @ 1.992GHz
 Y$$.    `.`"Y$$$$P"'         Memory: 47MiB / 3936MiB
 `$$b      "-.__
  `Y$$
   `Y$$.
     `$$b.
       `Y$$b.
          `"Y$b._
              `"""
```

לא ביצעתי שום שינויים מרכזיים בקרנל (מלבד שינוי ה-label) בשביל להשאיר את המאמר פשוט ומובן ככול הניתן אבל מכאן ואילך אתם יכולים לקחת את הכל ולשנות את מה שאתם רוצים.

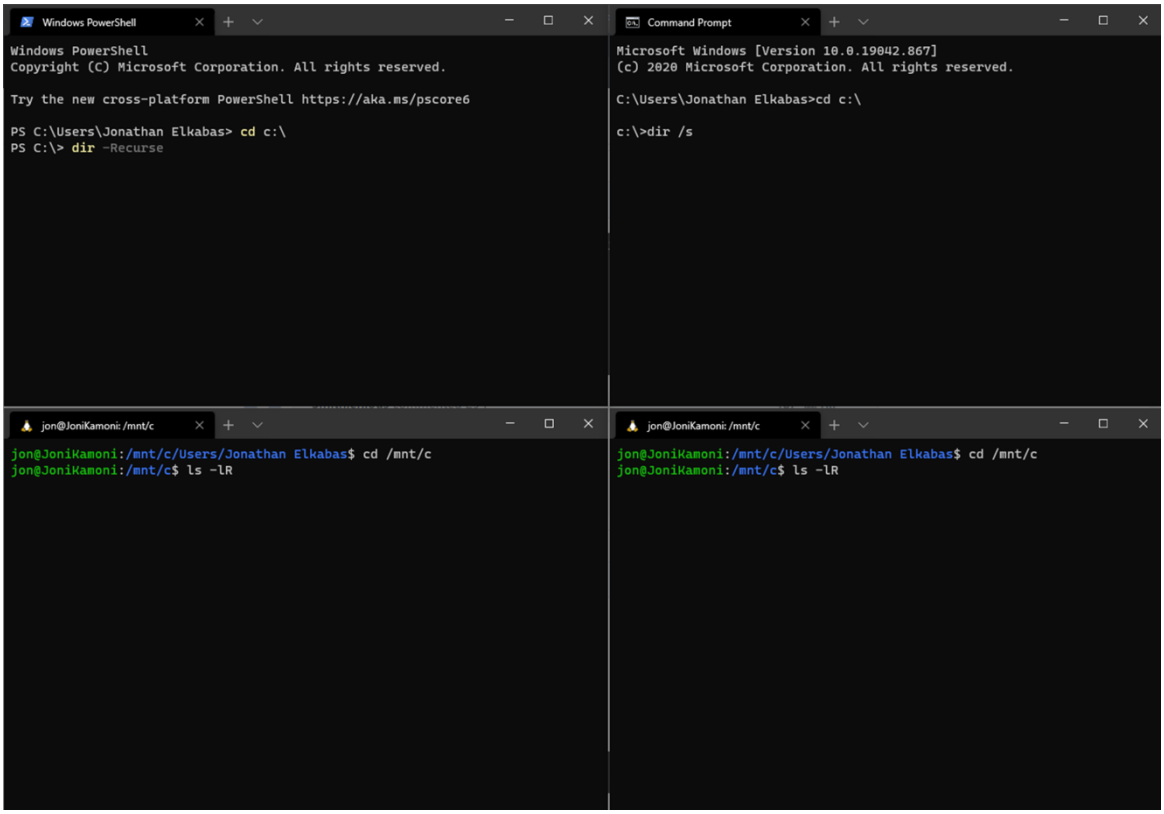
תזכרו שבזכות יכולת ה-CPU virtualization כל מה שרץ ב-WSL, כולל הקרנל שלנו, עכשיו מדבר ישירות עם החומרה של המחשב דרך ה-hypervisor ולא דרך הקרנל של Windows. **איך זה יתכן אבל?**

האם זה הגיוני בכלל ש-Windows ולינוקס ירוצו natively על אותה חומרה במקביל?

למעשה, מייקרוסופט מבטיחים לנו שכל עוד Hyper-V רץ בשכבה התחתונה ביותר זה אכן אפשרי להריץ מספר מערכות הפעלה שונות בצורה אפקטיבית ב-native mode בו"ז. כולן יכולות לגשת לחומרה, להתקנים ולזיכרון שהוקצה להם ב-hypervisor session; הן 'side by side' על אותה מכונה. זוכרים את האיור החמוד מעמוד 4?

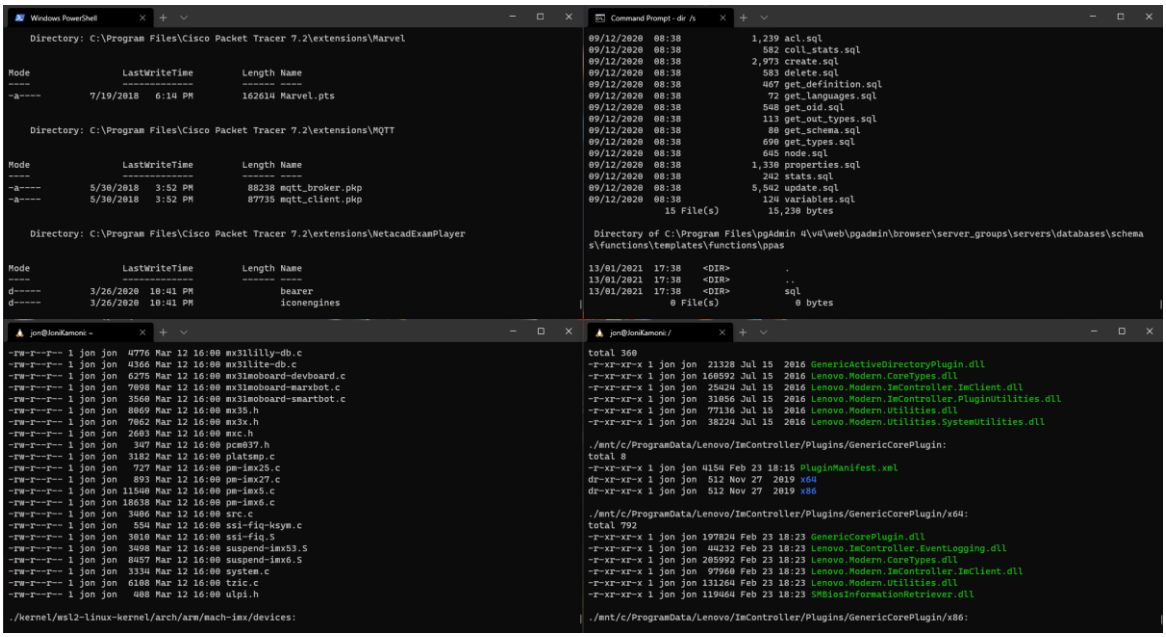
נבדוק את הטענה הזו.

בתמונה הבאה אתם יכולים לראות שפתחתי מספר סביבות שונות - PowerShell, Command Prompt ושתי Linux shells - אחת מ-Ubuntu ואחת של Debian. כולן רצות על ה-hypervisor:



אקנפג את ה-linux shell לרוץ מעל ה-mountpoint של מערכת הקבצים של Windows על מנת שאוכל להריץ בכל הטרמינלים במקביל את כל התוכן של ה-harddrive.

משימה מטופשת אבל היא תעשה את העובדה. בזמן ריצה של הטרמינלים כל הסיפור יראה בערך ככה:



נפתח את Windwos task manager (האהוב) ונראה באמצעותו תמונה חלקית של מה באמת קורה כאשר אנחנו מתחילים להריץ אותם אחד אחרי השני:

Name	Status	CPU	Memory	Disk	Network	GPU	GPU engine
Windows Terminal (13)		30.7%	120.3 MB	0 MB/s	0 Mbps	5.7%	GPU 0 - 3D
Windows PowerShell		19.6%	24.8 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		9.1%	0.9 MB	0 MB/s	0 Mbps	0%	
Windows PowerShell		2.0%	22.0 MB	0 MB/s	0 Mbps	5.7%	GPU 0 - 3D
jon@JoniKamoni: /mnt/c		0%	22.8 MB	0 MB/s	0 Mbps	0%	
jon@JoniKamoni: /mnt/c		0%	22.0 MB	0 MB/s	0 Mbps	0%	
Command Prompt - dir /s		0%	21.8 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	0.9 MB	0 MB/s	0 Mbps	0%	
Windows Command Processor		0%	0.7 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	0.9 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	1.0 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.7 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.7 MB	0 MB/s	0 Mbps	0%	
Runtime Broker		0%	1.0 MB	0 MB/s	0 Mbps	0%	

בצילום למעלה אפשר לראות שרק הטרמינל של PS רץ. נוסיף גם את ה-Command Prompt:

Name	Status	77% CPU	44% Memory	2% Disk	0% Network	26% GPU	GPU engine
Windows Terminal (13)		67.9%	124.1 MB	0.1 MB/s	0 Mbps	6.8%	GPU 0 - 3D
Windows PowerShell		22.1%	25.1 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		21.9%	0.9 MB	0 MB/s	0 Mbps	0%	
Windows Command Processor		12.2%	1.2 MB	0.1 MB/s	0 Mbps	0%	
OpenConsole.exe		6.1%	0.9 MB	0 MB/s	0 Mbps	0%	
Command Prompt - dir /s		4.6%	24.6 MB	0 MB/s	0 Mbps	6.7%	GPU 0 - 3D
Windows PowerShell		1.1%	22.4 MB	0 MB/s	0 Mbps	6.8%	GPU 0 - 3D
jon@JoniKamoni: /mnt/c		0%	22.0 MB	0 MB/s	0 Mbps	0%	
jon@JoniKamoni: /mnt/c		0%	22.8 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	0.9 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	1.0 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.7 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.7 MB	0 MB/s	0 Mbps	0%	
Runtime Broker		0%	1.0 MB	0 MB/s	0 Mbps	0%	
System interrupts		3.7%	0 MB	0 MB/s	0 Mbps	0%	

עם הוספת Ubuntu (בתמונה למטה) אפשר לראות שהתווספו משימות של "COM Surrogate" ו-"Vmmem":

Name	Status	100% CPU	43% Memory	2% Disk	0% Network	40% GPU	GPU engine
Windows Terminal (13)		59.5%	147.6 MB	0 MB/s	0 Mbps	5.9%	GPU 0 - 3D
Command Prompt - dir /s		12.8%	28.1 MB	0 MB/s	0 Mbps	5.2%	GPU 0 - 3D
Windows PowerShell		12.6%	25.2 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		12.3%	1.5 MB	0 MB/s	0 Mbps	0%	
Windows Command Processor		10.7%	1.3 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		6.7%	1.5 MB	0 MB/s	0 Mbps	0%	
jon@JoniKamoni: /mnt/c		1.8%	29.5 MB	0 MB/s	0 Mbps	4.5%	GPU 0 - 3D
OpenConsole.exe		1.5%	1.6 MB	0 MB/s	0 Mbps	0%	
Windows PowerShell		1.2%	25.9 MB	0 MB/s	0 Mbps	5.9%	GPU 0 - 3D
Microsoft Windows Subsystem...		0%	0.9 MB	0 MB/s	0 Mbps	0%	
jon@JoniKamoni: /mnt/c		0%	28.1 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		0%	1.5 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.9 MB	0 MB/s	0 Mbps	0%	
Runtime Broker		0%	1.3 MB	0 MB/s	0 Mbps	0%	
COM Surrogate		20.4%	2.0 MB	0 MB/s	0 Mbps	0%	
Vmmem		6.5%	442.0 MB	0 MB/s	0 Mbps	0%	
Desktop Window Manager		4.1%	105.2 MB	0 MB/s	0 Mbps	24.0%	GPU 0 - 3D
System		3.9%	0.1 MB	1.0 MB/s	0 Mbps	0.1%	GPU 0 - 3D
System interrupts		3.5%	0 MB	0 MB/s	0 Mbps	0%	

תהליך Vmmem מייצג את המשאבים שבשימוש על ידי מכונה וירטואלית (ליתר דיוק, מה שבאמת מיוצג אלו המשאבים שבשימוש על ידי מכונה וירטואלית **אחרת** מכיוון שגם Windows עצמה רצה בתוך VM מעל ה-Hypervisor). תהליך COM Surrogate (מייצג את dllhost.exe) שהדרך הכי טובה לתאר את מה שהוא עושה היא פשוט לצטט את מייקרוסופט:

> *"the COM Surrogate is the I don't feel good about this code, so I'm going to ask COM to host it in another process. That way, if it crashes, it's the COM Surrogate sacrificial process that crashes instead of me process."*

בקיצור, הוא אחראי לביצוע host לקבצי dll בסיכון גבוה לקריסה ומכיוון שאנחנו משתמשים ב-WSL דרך hyper-v הנאמן שלנו, בערך מיליון בינאריים חייבים לרוץ בשביל לתמוך בכל התשתית.

נוסיף גם את ה-shell של Debian לכל הסיפור ונראה שבהתאם להוספתו כל תהליך ירד בכמות המשאבים שהוא צורך (עקב האופטימזציה שהמעבד נדרש לעשות) ומכיוון שהוא מבוטא ב-subsystem נפרד אז גם במקרה הנ"ל 2 התהליכים של Vmmem ו-COM Surrogate עלו בצריכת המשאבים:

Name	Status	100% CPU	43% Memory	1% Disk	0% Network	48% GPU	GPU engine
Windows Terminal (13)		49.3%	149.5 MB	0 MB/s	0 Mbps	5.4%	GPU 0 - 3D
Windows PowerShell		12.7%	25.0 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		10.7%	1.5 MB	0 MB/s	0 Mbps	0%	
Windows Command Processor		9.6%	1.4 MB	0 MB/s	0 Mbps	0%	
Command Prompt - dir /s		7.1%	28.3 MB	0 MB/s	0 Mbps	4.7%	GPU 0 - 3D
OpenConsole.exe		4.8%	1.5 MB	0 MB/s	0 Mbps	0%	
Windows PowerShell		1.1%	26.3 MB	0 MB/s	0 Mbps	5.4%	GPU 0 - 3D
OpenConsole.exe		1.0%	1.6 MB	0 MB/s	0 Mbps	0%	
jon@JoniKamoni: /mnt/c		1.0%	29.9 MB	0 MB/s	0 Mbps	5.2%	GPU 0 - 3D
jon@JoniKamoni: /mnt/c		0.8%	29.2 MB	0 MB/s	0 Mbps	5.4%	GPU 0 - 3D
OpenConsole.exe		0.5%	1.6 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0.1%	1.0 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	1.0 MB	0 MB/s	0 Mbps	0%	
Runtime Broker		0%	1.3 MB	0 MB/s	0 Mbps	0%	
COM Surrogate		24.8%	2.1 MB	0 MB/s	0 Mbps	0%	
Vmmem		8.2%	442.0 MB	0 MB/s	0 Mbps	0%	
System		7.6%	0.1 MB	0.6 MB/s	0 Mbps	0.1%	GPU 0 - 3D
Desktop Window Manager		5.4%	101.9 MB	0 MB/s	0 Mbps	27.0%	GPU 0 - 3D
System interrupts		2.7%	0 MB	0 MB/s	0 Mbps	0%	

רק לשם השוואה (והבנה של הרקע התיאורטי), **הנמכתי את WSL לגרסה 1** וחזרתי על הרעיון של מיפוי הכונן באמצעות שלוש subsystems שונות (ארבעת הטרמינלים כמו מקודם). כעת אפשר לראות שבמקום שיפתחו לנו תהליכים של "COM Surrogate" ו-"Vmmem" נפתחו לנו שני תהליכים של "ls" שהם למעשה פשוט תרגום של ה-syscall וכל אחד מהם רץ כ-process נפרד:

Name	Status	100% CPU	48% Memory	2% Disk	0% Network	51% GPU	GPU engine
Windows Terminal (15)		59.2%	97.2 MB	1.7 MB/s	0 Mbps	5.2%	GPU 0 - 3D
Windows PowerShell		11.8%	15.0 MB	0.1 MB/s	0 Mbps	0%	
OpenConsole.exe		10.2%	0.5 MB	0 MB/s	0 Mbps	0%	
Windows Command Processor		8.8%	1.0 MB	1.7 MB/s	0 Mbps	0%	
OpenConsole.exe		5.9%	0.5 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		5.7%	0.5 MB	0 MB/s	0 Mbps	0%	
OpenConsole.exe		5.3%	0.5 MB	0 MB/s	0 Mbps	0%	
Command Prompt		4.0%	20.3 MB	0 MB/s	0 Mbps	5.2%	GPU 0 - 3D
jon@JoniKamoni: /		3.4%	19.2 MB	0 MB/s	0 Mbps	4.7%	GPU 0 - 3D
Windows PowerShell		2.1%	19.1 MB	0 MB/s	0 Mbps	5.1%	GPU 0 - 3D
jon@JoniKamoni: ~		2.1%	18.4 MB	0 MB/s	0 Mbps	5.1%	GPU 0 - 3D
Console Window Host		0%	0.1 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.4 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.4 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Subsystem...		0%	0.4 MB	0 MB/s	0 Mbps	0%	
Runtime Broker		0%	1.0 MB	0 MB/s	0 Mbps	0%	
ls		14.4%	0.5 MB	0.1 MB/s	0 Mbps	0%	
ls		13.6%	0.8 MB	0 MB/s	0 Mbps	0%	

זו גם ההוכחה (ברמה שיטחית ביותר) מדוע WSL1 פחות מהיר משמעותית מ-WSL2; הוא פותח process נפרד להרצת הפקודה (לאחר תירגום) במקום להקצות זיכרון וירטואלי ולבצע עליו ישירות את הפקודה.

בשביל מה אני צריך את זה?

"שמע אמרת פה הרבה דברים מעניינים אבל עדיין לא קנית אותי. למה שאני אצטרך להריץ Subsystem נוסף ב-WSL? מה זה נותן לי? **איפה ה-benefit בכל הסיפור?"**

שאלות לגיטימיות. הדרך הכי טובה להראות את היתרונות של תשתית WSL היא באמצעות **דוגמאות פרקטיות**:

הרצת subsystem לינוקסי כדוגמת Ubuntu תחת WSL מאפשר להריץ *במסך אחד* אפליקציות שלא היינו מסוגלים להריץ תחת Windows בעוד *שבמסך השני* נוכל להריץ אפליקציות Windows-יות שאין בלינוקס וכך לשלב את סביבות העבודה שלנו לאחת ולנהל אותה ממקום אחד.

אם אתם תוכניתנים ואתם עובדים על אפליקציה שהצד שרת שלה מבוסס לינוקס ואתם צריכים לבצע עריכה גראפית, ניתן למשל להריץ Photoshop או Premier בצד ה-Windows-י בשביל התצורה של אובייקט כלשהו בצד ה-front end בעוד שאפשר לערוך את קוד השרת ב-vim (או כל עורך אחר) כך שיתאים לשינויים. נשמע טיפה מוזר לחלקכם אבל זה בהחלט פיצ'ר מאוד שימושי ל- fullstack developers. כמו כן, חשוב מאוד לציין שאפשר לקשר את WSL ל-VScode ואז כל הסיפור נעשה משמעותית יותר פשוט.

אם אתם אנשי DevOps שימוש פרקטי יכול להיות למשל אם אתם רוצים לבדוק CI/CD pipeline (Continuous Integration & Continuous Delivery) ומעוניינים לבדוק אותו לוקאלית (על הלפטופ) לפני שאתם מעלים אותו לענן. אפשר ליצור instance של Ubuntu ב-WSL ולאחר שאתם מוודאים שה-pipeline עובד כשורה בצורה לוקאלית (עם אלו כלים ותוכנות שאתם מעדיפים) אתם יכולים לשים אותו ב- Docker container ולעשות לו push אל הענן (למשל Azure) ושם הוא ירוץ על מכונת VM של Ubuntu בסביבת פרודקשן. זה נכון שאפשר לעשות הכל גם דרך אפליקציית VM סטנדרטית כמו Virtualbox/Vmware אבל WSL משתמש משמעותית בפחות משאבים (CPU, זיכרון ואחסון) ובעל גישה לכל הקבצים ב-Windows (וזה שימושי במיוחד אם הקוד שלכם ו\או סביבת ה-testing יושבים ב-Windows).

* * *

בכל מקרה, אני לא יודע מה איתכם אבל אני 'לא מחבב במיוחד' את ה-shells של Windows. בין אם זה המיליון משתנים שצריך לתת ב-Powershell או זה שחסרים מלא פיצ'רים ב-cmd הרגיל. העובדה שאפשר להריץ grep, awk, diff, ssh על Windows מספיקה מבחינתי בשביל להצדיק את ה'מעבר' ולהוריד Image של Ubuntu ששוקל 450MB (יש גם Debian ששוקל 78MB [!]).

אישית, אני עדיין משתמש גם ב-WSL וגם ב-Vmware Workstation Pro מהסיבה הפשוטה שלכל אחד מהשניים יש את היתרונות שלו (בהצלחה בביצוע image cloning / snapshot manipulation וקינפוג.

מאפייני רשת של VMs דרך WSL). בסופו של יום WSL הוא אחלה כלי ששווה לכל אחד להכיר ולהוסיף לארגז הכלים שלו.

סיכום

בנינו קרנל לינוקסי משלנו והרצנו אותו על גבי תשתית WSL. מכאן ניתן לשפץ ולערוך את הקרנל כך שיכיל בינאריים ומודלים בהתאם לדרישות ולרצונות שלנו. בין זה תמיכה בחומרה וחקירת פיצ'רים או בין אם זו מערכת קבצים חדשה שבא לנו לנסות.

הצלחנו להריץ מספר subsystems במקביל ודיברנו על היתרונות בכל הסיפור. אני מקווה שכעת גם אתם רואים את הפוטנציאל שטמון בפיצ'ר הנחמד הזה שנקרא WSL. אתם יכולים להריץ Windows Desktop ולקבל את ה-user experience שכולם משום מה נורא אוהבים מבלי לוותר על כל מה שניתן לעשות עם הקרנל של לינוקס - וזה כולל בתוכו את כל מגוון ה-flavors ו-distributions. אפשר להשתמש ב-**Windows** וב-desktop shell בתור ה-host בשביל כל האחרים.

בקיצור - אפשר לאכול את העוגה ולהשאיר (את רובה) שלמה. ☺

על הכותב

יהונתן אלקבס, בן 26, חוקר אבטחת מידע בחברה לא קטנה ולא פרטית. חובב קקטוסים, סודה ומכור לקפה.