**שקף 2**

אז בואו נתחיל ממשהו מאוד טריוויאלי – מה זה IO . אז זה בעצם ראשי תיבות של Input or output operation .

בהגדרה המילונית של IO היא התקשורת בין information processing system , לדוגמא – מחשב , והעולם החיצוני – שהוא בדרך כלל עוד information processing sytem .

Input - signals or data שמתקבלים למערכת ו-outputs הם signals or data שהמערכת שולחת.

עכבר ומקלדת למשל הם Input devices למערכת , ומקלדת לדוגמא, היא output devices .

אבל ספציפית לשרתים , כאשר אנחנו מדברים על פעולות IO אנחנו מדברים בעיקר על תקשורת עם מערכות חיצוניות, לדוגמא הרצת שאילתה מול ה-DB , קריאה מקובץ, שליחת message ל-queue , שליחת message לשרת אחר וכו'. כל דבר כזה שאתם בעצם יכולים לחשוב עליו.

פעולות IO הן יותר איטיות ב-order of magnitude מפעולות שעושים של עיבוד זיכרון.

**שקף 3 :**

אז ברוב המערכות שאנחנו מכירים, כל פעולות ה-IO שהזכרתי עכשיו הן blocking . זאת אומרת , שה-execution של ה-program שמבצעת את פעולת ה-io בעצם יפסיק ויחכה לקריאת ה-IO שתסתיים ותחזיר את ה-result . רק אחרי שה-result חוזר אז התוכנית תמשיך. זה בעצם מאוד make sense כי כמעט תמיד צריך את התוצאה של קריאת ה-io בחישוב וכדי לבצע לוגיקה.

**שקף 4:**

אז בעצם, כשיש לנו single user , זה נשמע מאוד הגיוני. אבל בואו נדבר על קצת המזלג מה קורה בסביבה של multi users - client server .

בשפות 'מסורתיות' , כמו לדוגמא Java ו-c# , בעצם פותחים thread או process חדש כדי לטפל ב-request .

אז אם יש לנו בעצם 100 parallel requests השרת יצטרך 100 processes או threads כדי לטפל בהם. בואו נגיד שהשרת שלנו פותח Thread חדש – אנחנו עדיין נצטרך 100 threads . זה אומר שאם 100 threads מחכים ל-Result של פעולת DB ומגיע Request חדש – אז יווצר thread חדש .

**שקף 5:**

אז קודם כל, פתיחת Thread צורכת עוד זיכרון ועוד משאבי CPU . בנוסף, פתיחת מספר רב של threads יכול לגרום למערכת עמוסה 'לבלות' זמן רב ב-thread scheduling ו-context switching – מעבר בין משימות של threads שונים, שמוסיפים לזמן התגובה של השרת – יחזיר response יותר לאט וגורמים למגבלות על ה-scalability .

**שקף 6:**

אז התחילו לחשוב על גישה חלופית – non blocking IO .

אז מה זה בעצם אומר ?

אז בעצם כל task שתלוי ב-IO שחייב להסתיים עדין צריך לחכות , אבל בזמן הזה , ה-Thread יבצע porcessings אחרים שלא תלויים ב-IO , והם אכן ימשיכו וירוצו.

מה שקורה מאחורי הקלעים הוא שה-IO request הוא queued מייד והפונקציה חוזרת , וכך אפשר לבצע פעולות נוספות עד שפעולת ה-IO חוזרת.

**שקף 7:**

NodeJS הפלטפורמה הוא single threaded עם event loop .זה בעצם אומר שה-event loop של Node רץ על thread אחד.

מה שבעצם קורה זה ב-high level שכל connection חדש שיוצרים לשרת בעצם קורא ל-callback function . ה-call back function מטפלת ב-Requests בעזרת מודל שנקרא non blocking io – זה בעצם אומר שבזמן שמחכים לפעולת IO שתחזור הthread בינתיים מטפל ב-events אחרים ומספיק בעצם לעשות המון דברים אחרים.

**שקף 8:**

ה-event loop הוא מה שמאפשר ל-NodeJS לבצע non blocking io operations , למרות העובדה ש-Javacsript הוא single threaded – על ידי offloading של ה-operations ל-system kernel בכל פעם שאפשר.

רוב ה-kernels המודרניים הם multi threaded , ולכן הם יכולים לטפל ב-multiple operations ולבצע אותן ברקע. כשאחת מהפעולות מסתיימת, ה-kernel אומר ל-NodeJS כך שה-callback המתאים יכנס לתור של ה-events ולבסוף יתבצע.

**שקף 9:**

אז כאן יש דיאגרמה של הארכיטקטורה של NodeJS . אני בעיקר רוצה שנתמקד בצד שמאל של הדיאגרמה . שימו לב מה קורה כאן :

יש כאן את ה-queue של ה-events שבו אנחנו מקבלים את ה-events שצריכים להתבצע.

יש כאן event loop שרץ על Thread אחד. הוא בעצם מוציא את ה-events ומבצע אותם.

Blocking operations בעצם הן offload ל-worker pool .

הכל ממומש על LibUV .

**שקף 10:**

אז כשאפליקציית Node מתחילה , קודם כל משלימים initialization phase , require למודולים ו-registering callbacks עבור events .

לאחר מכן Node נכנס ל-event loop , ומגיב לincoming client requests על ידי ביצוע של ה-approapriate callbacks . ה-callback הספציפי שמטפל ב-Incoming client מבוצע באופן סינכרוני, אבל יכול רשום asynchronous requests כדי להמשיך את ה-processing אחרי שבקשה הא-סינכרונית מסתיימת. גם ה-callbacks עבור ה-async requests יהיו executed ב-event loop .

בנוסף, ה-event loop ימלא את ה-non blocking asynchronous requests , על ידי ה-callbacks שלהן שחזרו .

אז לסיכום : ה-event loop מבצע JavaScript callbacks שנרשמו עבור events , ובנוסף אחראי למלא non blocking asynchronous requests כמו network ו-IO /

**שקף 11:**

אז ה-worker pool הוא בעצם thread pool שאחראי על לבצע פעולות כבדות. פעולות כבדות יכולות להיות פעולות IO או פעולות שהן CPU intensive . זהו pool שחסום ב-capacity קבוע .

ה-Worker pool הוא Implemented ב-libuv , שבעצם חושף API גנרי להגשת tasks .

בעצם מודולים ב-Node שמשתמשים ב-worker pool מגישים את ה-task לpool דרך ה-API של libuv .

כדי לעשות את זה , חייבים להתשתמש ב-c++ add on .

**שקף 12:**

ניקח כמה דוגמאות למודולים שעובדים עם ה-worker pool :

דוגמאות למודולים שהם IO intensive הם : DNS – לדוגמא פעולות כמו DNS lookup , ה-file system API – כולו משתמש ב-worker pool ,חוץ מפעולות שהן סינכרוניות

CPU intensive : מודול crypto משתמש ב-worker pool כדי לבצע פעולות שהן CPU intensive . יש לנו כמה דוגמאות נבחרות כאן.

**שקף 13:**

אז בואו נדבר על איך שה-event loop וה-worker pool ממומשים.

**שקף 14:**

אז רק כדי שניכנס למימוש מאוד מאוד ב-high level של ה-event loop – הוא לא מתחזק queue מאחורי הקלעים. הוא מחזיק collection של file descriptors , והוא מבקש ממערכת ההפעלה לעשות עליהם מוניטור , בעזרת שיטות כמו epoll (linux) , kqueue (OSX) , event ports(solaris) ועוד. ה-file descriptors בעצם מתורגמים ל-network sockets , ל-files ספציפיים וכו. כשמערכת ההפעלה "אומרת" שאחד מה-file descriptors מוכן , ה-event loop מתרגם את זה לappropriate event ומבצע Invoke לcallback המתאים.

**שקף 15:**

בניגוד ל-event loop , ה-worker pool כן מתחזק queue אמיתי , שבו יש משימות שצריך לבצע להן execution . ה-worker מוציא task מה-queue ועובד על ה-task , וכאשר הוא מסיים הוא מרים event מתאים ל-event loop .

**שקף 16:**

אני רק רוצה להגיד כמה מילים על libuv – זו בעצם ספרייה שכתובה ב-C , היא cross platform , שמספקת תמיכה בasync IO שמבוסס על event loop . תומכת במנגנונים כמו epoll (linux) , kqueue (OSX) , event ports(solaris).

**שקף 17:**

אז עכשיו בואו נדבר על סדר הפעולות של ה-event loop – או – באיזה סדר הפעולות מתבצעות .

**שקף 18:**

אז השלב הראשון הוא timers – שבו יהיה execution של ה-callbacks שהם scheduled על ידי setTimeout ו-setInterval .

השלב השני הוא IO callbacks – מבצע כמעט את כל ה-callbacks חוץ מה-timers , setImmediate ו-close callbacks

השלב השלישי הוא שלב שנקרא idle ו-preapre – שנעשה בו שימוש רק Internally .

השלב הרביעי הוא שלב שנקרא poll – בעצם מושך IO events חדשים , node יחסום כאן כאשר צריך.

השלב החמישי הוא check – כאן יש invoke רק ל-callbacks שנוצרו על ידי setImmediate .

השלב האחרון הוא close callbacks - לדוגמא callbacks שנוצרו על ידי socket on close events .

**שקף 19:**

לכל Phase יש fifo queue של callbacks לעשות להם execute . כשהevent loop נכנסת ל-Phase מסויים, היא מבצעת את הפעולות עד שה-queue היגיע ל-limit שלו. כאשר החגענו ל-limit , ה-event loop יעבור לשלב הבא.

כאשר כל phase בעצם מסתיים , עוברים ל-phase הבא, ומתחילים לבצע את ה-callbacks שנמצאות ב-queue של ה-phase הבא.

**שקף 20 :**

Timers קובעים את ה-threashold שאחריו callback שהמשתמש סיפק יהיה executed . לא מדובר כאן על זמן מדוייק שבו ה-callback יתבצע . timers callbacks ירוצו הכי מוקדם שהם יכולים להיות scheduled אחרי שהspecified amount של הזמן שנקבעה להם עברה. הם יכולים להתאחר על ידי ה-scheduling של ה-Operating system או על ידי ריצה של Operations אחרים.

אני רוצה להגיד במאמר מוסגר שה-poll phase קובע מתי ה-cllabcaks יהיו scheduled .

**שקף 21:**

בשלב זה יתבצעו Callbacks עבור כמה system operations כמו Types של TCP errors . לדוגמא , אם TCP socket תקבל Error של connection refused , כאשר מנסים להתחבר, חלק ממערכות ההפעלה רוצות לחכות עד שיהיה report ל-error הזה . ה-callback הזה יהיה queued ו-executed ב-IO callback phase .

**שקף 22 :**

בשלב ה-Poll יש שתי פונקציות עיקריות :

1. ביצוע callbacks ל-timers שה-threshold שלהם elapsed
2. ביצוע events שב-poll queue .

כשה-event loop נכנסת ל-poll phase ואין scheduled timers , יקרו הדברים הבאים :

* אם ה-poll queue לא ריק , ה-event loop יעבור על ה-callbacks ב-queue ויבצע אותם עד שה-limit של ה-queue היגיע.
* אם ה-poll queue ריק, יקרו אחד מ2 הדברים הבאים :
  + אם נקבעו callbacks של setImmediate , ה-event loop תסיים את ה-Poll phase ותעבור ל-check phase כדי לבצע callbacks שנקבעו על ידי setImmediate .
  + אם לא נקבעו callbacks של setImmediate , ה-event loop תחכה עד שיכנסו callbacks חדשים ותבצע אותם מיידית.

כאשר ה-poll queue ריק ה-event loop תבדוק אם ישנם timers שעבורם ה-time threshold עברו. אם ישנם כאלה, ה-event loop יחזור ל-timers phase לבצע את הcallbacks ורק אז ימשיך הלאה.

**שקף 23:**

הסבר קצר לגביי ה check phase :

בעצם מאפשר לעשות execute ל-callbacks מיידי על ידי setImmediate , מיד לאחר שה-poll phase נגמר . אם ה-poll phase נהיה idle וה-callbacks יהיו queued עם setImmediate , ה-event loop תמשיך ל-check phase במקום לחכות ותסיים את ה-poll phase .

הסבר לגביי ה-close phase :

מיועד ל-events עבור סגירה של sockets לדוגמא שנסגרו באופן לא צפוי.

**שקף 24:**

בטח שמתם לב שלא הזכרתי את process.nextTick . זה בגלל שהוא לא חלק מה-event loop . process.nextTick יהיה processed אחרי שה-current operation תסתיים, ולא משנה באיזה שלב. לכן , צריך להיזהר בשימוש שלו – כי הוא יכול לגרום ל-starvation . ברגע שקוראים לו – כל ה-callbacks שמעבירים לו יכולים להיות resolved .ואז ה-phases הבאים של ה-event loop מחכים.

**שקף 25:**

אז למה בעצם Node יכול to scale בצורה טובה ?

**שקף 26:**

אז יש לנו מספר מאוד מצומצם של threads – אחד לevent loop ועוד מספר קבוע עבור ה-workers threads . לכן , במקום לפתוח thread נוסף לכל request , בעזרת מספר מצומצם של threads אנחנו משרתים יותר בקשות , ויכולים להתרכז בעבודה שאנחנו צריכים לעשות במקום בלבזבז משאבי CPU וזיברון על פתיחת Threads נוספים. בעצם כל thread לוקח מאיתנו overhead גדול כזה. ומביא איתו גם בנוסף overhead של context switching ו-thread scheduling .

**שקף 27:**

אז בעצם – blocking thread יכול להיות ברמת ה-event loop וברמת ה-workers pool . ברמת ה-event loop – אלגןריתם שהוא CPU intensive לדוגמא. ברגע ש-thread חסום הוא לא יכול לטפל ב-requests של clients אחרים וכל השרת עובד ב-throughput הרבה יותר נמוך.

**שקף 28:**

לכן מה שחשוב כאן הוא לבנות את השרת שלכם באופן חכם – כשכלל האצבע החשוב ביותר הוא –

Node is fast when the work associated with each client at any given time is "small"