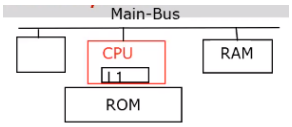
מערכות הפעלה – סיכום.

הרצאה 1:

התפקיד המינימליסטי של מערכות הפעלה זה לאתחל את החומרה.

איך זה עובד ? על ה CPU יש L1 (לאודר) שהוא רכיב זיכרון קטן שמקשר את המעבד (CPU) לזיכרון הROM וכך הCPU מתחיל לאתחל.



כדי ששני הרכיבים יתקשרו ביניהם יש את ערוץ התקשורת bus שמקשר ביניהם.

הבסיס למערכות הפעלה גדולות כמו win או linux אנחנו חייבים BIOS/bootloader שהיא מערכת הפעלה מינימליסטית שרק מאתחל את החומרה.

ניהול משאבים – על מערכת ההפעלה לנהל את המשאבים (כגון זיכרון, תהליכים, מקום על הדיסק ,כמות הליבות בCPU וכדומה).

ליבה – יחידת חומרה שיכולה להריץ תוכנה.

CPU management – ניהול הליבות בתוך הCPU , במעבד עצמו יש scheduler אשר מתזמן את חלוקת התהליכים לליבות, כאשר יש גם פעולות שלא חייבים ליבה כדי לרוץ כמו תהליכים שקשורים לכרטיס רשת וכדומה.

אז ברגע שמתבצע כתיבה או קריאה (I\O) אזי אנחנו עובדים מול הdivice ולא מול ה CPU, למשל כאשר אנחנו עובדים מול דיסק אז הדיסק הוא זה שעובד ולא הCPU (שרק מניע את התהליך).

Memory management - ישנם כמה סוגי זיכרון, הזיכרון שהגישה אליו היא הכי מהירה נקרא registers ובו מועבר הdata.

בCPU יש את סוגי הזיכרון הבאים registers , cache(L1-L4), main memory (ram) , second memory .

Interrupts ,interrupts handlers –אות חשמלי שמתקבל בCPU , כאשר לCPU יש עדיפות לטפל בinterrupts שמגיעים כאשר תהליך הטיפול קודם הוא מאשר את קבלת המידע ואז מפעיל Interrupt handle והוא דוחה את כל מה שהוא עושה כרגע ומטפל רק בinterrupt.

Kernel /user space – פעם כל הפיתוחים והאפליקציות היו באותו מרחב כתובות דבר שהיה פוגע בפיתוח ובתפקוד של ה CPU ושל שאר האפליקציות.

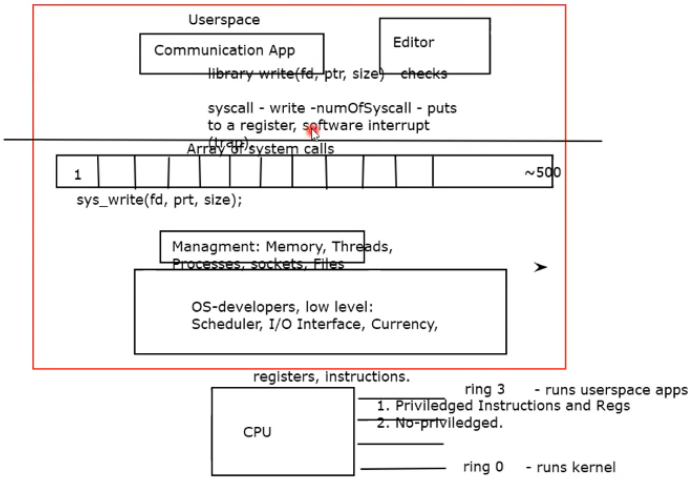
כיום לקחו את כל המרחב וחילקו אותו ל kernel ו user כאשר אותם מפתחי אפליקציות חסומים מלהגיע לאותם הכתובות, יצרו שני מרחבים שונים כאשר האפליקציות היו ב user place ואילו מערכות ההפעלה והתקשרת הקשורות יותר לחומרה יהיו בkernel place .

כאשר בשלב הבא גם חילקו את ה user place לשנים לProcess ואפליקציות.

ל CPU יש כמה סוגי גישות הנקראים ring כאשר למשל ring0 נותן לך גישה להריץ את הkernel ואילו ring3 ייתן לך גישה להריץ את האפליקציות הנמצאו בuser space.

בצורה זו נוצר סדר במערכות הפעלה פופולריות כמו ווינדוס לינוקס וכדומה.

כדי לעבור מ kernel ל user ישנו interrupt שמקשר בין השנים ולכן אנחנו בכלל יכולים לגשת לkernel ולקבל ממנו שירות (תקשורת וכו') תהליך ההתקשרות בין השנים נקרא system call.


Userspace
Communication App
Editor
library write(fd, ptr, size) checks
syscall - write -numOfSyscall - puts
to a register, software interrupt
(trap)
Array of system calls
1
~500
sys_write(fd, prt, size);
Managment: Memory, Threads,
Processes, sockets, Files
OS-developers, low level:
Scheduler, I/O Interface, Currency,
registers, instructions.
CPU
ring 3 - runs userspace apps
1. Priviledged Instructions and Regs
2. No-priviledged.
ring 0 - runs kernel

הרצאה 2 :

Multiprogrammin – כאשר CPU מריץ איזה שהיא פונקציה ובמקביל מגיע אליו איזשהו interrupt אזי כמובן שהCPU ייגש לטפל בו, כלומר הוא יצטרך לקפוץ לinterrupt ולכן נצטרך לבצע context switch - בעצם נחליף את המצביע שטיפל בפונקציה למצביע של ה interrupt .

תהליך ביצוע context switch – התהליך מתבצע על ידי זה שקודם נגיד לCPU לעצור ונשמור את התוכנית במחסנית של הthread ואז נטעון interrupt handler חדש ונגיד לCPU לרוץ ולאחר מכן נגיד לו לחזור למה ששמרנו.

– Thread and process

Thread – חוט, קו ריצה, תהליכון – ישות מיוחדת שיודעת לרוץ על CPU ,כלומר היא יודעת לקחת את המשאבים הרלוונטיים (stack, regristers – כי הוא צריך להריץ איזו פונקציה) ולטעון אותם בCPU .

כל thread מריץ תוכנה משלו והוא מאוד דומה ל interrupt handle אך בניגוד אליו הוא יכול להיחתך, כלומר גם אם ה thread לא סיים את הפונקציה אפשר לחתוך אותו.

אחד היתרונות הבולטים של threads זה המקביליות.

בכל ליבה יכול לרוץ thread אחת.

תהליך יצירת וריצת thread :

ניצור thread ע"י בקשה ב kernel space , יש תהליך שבודק אותו ולאחר הבדיקה הוא שואל האם יש לtread מה לעשות.

כאשר יש לו מה לעשות הוא נכנס ל ready של ה scheduler ולפי כל מיני אלגוריתמים הscheduler מחליט לתת ל thread ליבה ומשם הוא רץ עד אשר הוא מסיים את כל הפונקציה, ישנם עוד כמה סיבות שבו ה thread יוצא או מאבד את הליבה שלו : אם יש איזו שגיאה או לחילופין יש context switch למישהו אחר ,הסיבה הנוספת היא שה thread הגיע להמתנה או שהוא הניע תהליך I/O ולכן הוא נכנס למצב block והוא לא רץ כלל בליבה שלו.

Processes - בעבר לאחר חלוקה ל user ו kernel space עדין אפליקציה אחת יכולה לפגוע באפליקציה שניה כיוון ששניהם היו נמצאות בuser space ולכן פיתחו את הprocess שמגדיר לכל אחת מהאפליקציות מרחב זיכרון וירטואלי שבו אתה לא יכול לפגוע באפליקציה האחרת.

כאשר לכל process נותנים במינימום thread אחד שהוא באמת זה שרץ.

ה process מורכב מ:

1. Full VM space
2. At least one thread

3. Page table
4. Stack -> threads get their stack on stack of process
5. Heap
6. Data
7. Text/Code segments
8. Cmd-line args
9. Environment var

8. File/Socket descriptiors opened
9. Signals - settings

pid
ppid

permission
priviledges

הרצאה 3 :

ל process יש אופציה לפתוח בעצמו כמה threads בנוסף לאחד שלפחות יש לו בשביל לבצע דברים במקביל כלומר ה process יחלק את העבודה שלו לכמה threads וכל thread יבצע חלק מסויים.

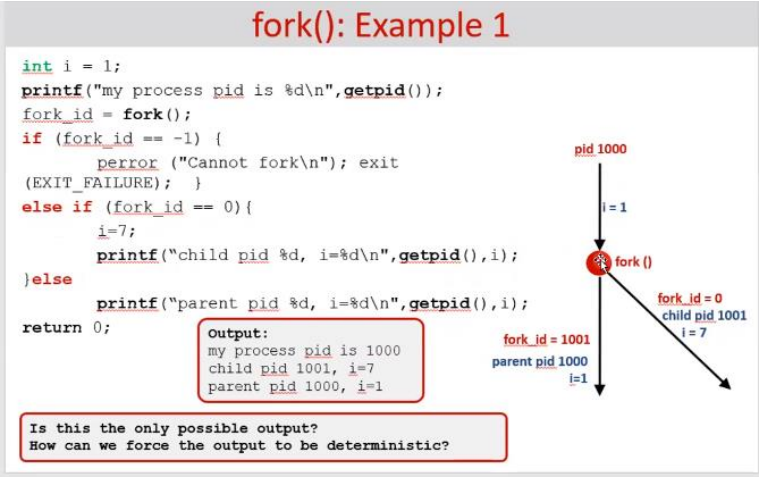
התנאי לביצוע תהליך כזה הוא קיום של כמה ליבות , בסוף תהליך המקביליות יהיה אלגוריתם שיעשה לכל הthreads merge ובצורה זו נתייעל.

לרוב threads יהיו שייכים לprocess מסוים אך ישנם גם threads אשר שייכים לkernel space שבו לא רצים כלל processes.

איך מייצרים process ? (fork ())

מהuser space ניתן ליצור Process ע"י פעולת fork () שלוקח איזשהו תהליך ומעתיק אותו בצורה מדויקת ,כאשר כל תהליך מקבל PID (process id).

קוד ליצירת process :



בדרך כלל ניצור שלושה מצבים :

ה if הראשון נועד למקרה בו לא הצלחנו ליצור את ה process כי המשאבים היו מוגבלים או שלל סיבות נוספות.

ה else if שבא לאחריו נועד עבוד הchild process - כלומר אנו מעתיקים את הקוד והוא רץ פעמיים, פעם כילד ופעם כאבא (הprocesses דומים אך שונים רק בPID), זה נועד עבור יצירת process נוסף ותפקידו של האבא לפקח על ה"ילד" שלו.

וה else האחרון מחזיר את הparent process.

פונקציית Exec –משפחה של פונקציות שמריצות תהליכים, מחליפות בין תהליכים.

סיום התהליך – wait :

ברגע שהתהליך מסתיים האבא צריך לעשות wait ואם הוא לא עושה את הפקודה הזו והבן מסיים את התהליך בלי שאבא מחכה ואוסף את הסטטוס שלו נוצר זומבי.

ובמקרה שהילד נפל במקרה שהאבא עושה wait אזי זה מבטיח לנו שהאבא יוליד process חדש במקומו, כלומר ה wait מבטיח לנו שהprocess של האבא ייתן לנו שירות כי הוא אחראי על כל מה שהוא מוליד.

: Threads vs. process

Threads – נועד לריצה עצמה על הCPU , process - מעטפת משאבים שכוללת יכולות ריצה (בגלל שהיא כוללת מינימום thread אחד) .

Threads vs. Processes

Processes	Threads
unique data	shared data
unique code	shared code
unique open I/O	shared open I/O
unique signal table*	shared signal table*
unique stack	unique stack
unique PC	unique PC
unique registers	unique registers
unique state	unique state
heavy context switch	light context switch

heap
env variable
command line
page table
TLB
stack for all threads

Priority of process
CPU-affinity mask

Unique stack borrowed on stack of the process

TLS/TSS - thread safe storage - thread-safe segment

CPU-affinity mask
Thread-priority

* Signal handlers must be shared among all threads of a multithreaded app. However, each thread must have its own mask of pending / blocked signals: use pthread_mask rather then sigprocmask.

הרצאה 4 :

IPC – inter process communication :

ה process צריך להעביר מידע ולהיות בקשר עם processes אחרים.

לכל process שנמצא ב user space יש גם מעין process קטן שנמצא גם ב kernel space שבו כל האינפורמציה על הprocess הוא נקרא PCB, כאשר גם לכל PCB כזה יש כמובן thread בשם TCB שמחזיק את כל האינפורמציה על הthread.

ברגע שיש לCPU איזו שגיאה הוא שולח לprocess signal עם הבעיה , כאשר כל התהליך של טיפול ב interrupts מתבצע אך ורק ב kernel space (כאשר בכלל לuser space מותר להפעיל interrupt אחד בשביל הsystem call).

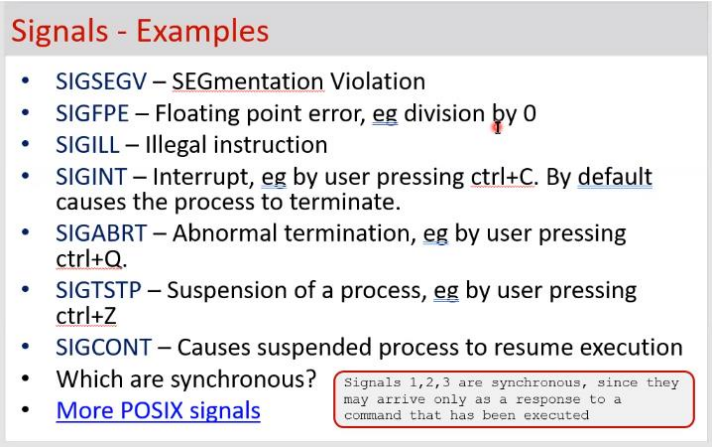
כלומר הsignals נועדו בכדי להמיר את ה Interrupts לאינפורמציה שתועבר לתהליכים שמטפלים בסיגנלים.

Signal :

סיגנל יכול להישלח לprocess על ידי process אחר או על ידי הkernel למעט שני הסיגנלים מהסוג SIGUSR1,SIGUSR2 .

שימוש נוסף לסיגנל זה לעשות blocking לחסום.

סוגי סיגנלים :



טיפול בסיגנלים – לסיגנלים יש שלושה אפשריות :

Default – SIG_DFLT – הפעלת פונקציית aboard שבודקת האם מותר לה לרשום קובץ core ואם כן היא רושמת לו הכל וכך נוצר קובץ core , כלומר הטיפול הוא לפי ההגדרה הדיפולטיבית של הsignal.

Ignore - SIG_ING – תהליך ה default לא יתבצע.

Handling – ביצוע של פונקציה שהמשתמש רשם בצורה לא שגרתית (במקום לקרוא לaboard).

קבלת סיגנלים :

הפונקציה signal () מקבלת את סוג הסיגנל ומקבלת גם את דרך ההתמודדות של ה process איתו, כלומר הפרמטר הראשון הוא איזה סיגנל אני מקבל והפרמטר השני מה אני אעשה כאשר אני מקבל אותו (דרך ההתמודדות שלי עם הסיגנל כמפורט לעיל) לדוגמא :

```
sighandler_t signal (int signum, sighandler_t handler)
```

הפונקציה sigpocmask () נותנת לנו לנהל את ה mask וגם את הblocking שהסיגנל יכול לעשות.

שליחת סיגנל :

ניתן לשלוח סיגנל ע"י הקלדה פשוטה במקלדת (Ctrl +c ,Ctrl -Q ,Ctrl-Z) וגם בשימוש עם המילה kill בצירוף סוג הסיגנל יחד עם הprocess id (היעד).

לדוגמא : kill – 9 <pid> .

דוגמא ליצירת סיגנל מסוג sig handler שבו המשתמש מדפיס את המסך בתגובה לסיגנל :

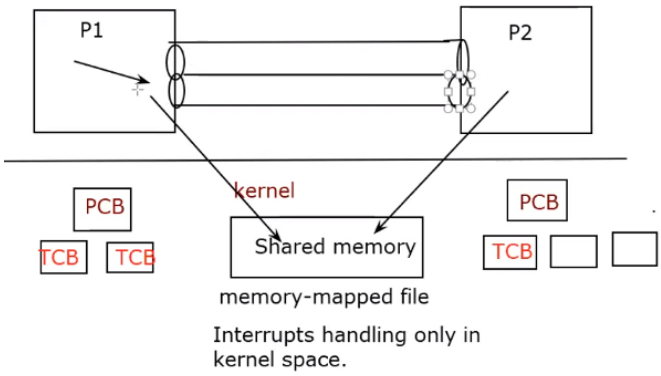
```
#include <signal.h>
#include <stdio.h>

void mySigTermHandler (int signum)
{
   fprintf (stderr,"Received signal %d\n", signum);
}

int main()
{
    signal(SIGTERM, mySigTermHanlder);
}
```

זיכרון משותף (כלים לIPC) :

נחזור לבעיה של תחילת השיעור בה יש שני תהליכים ב user place ולכל תהליך יש זיכרון משלו והם צרכים לתקשר ביניהם, הם יעשו זאת ע"י זיכרון משותף שיימצא ב kernel או ע"י memory -mapped file .

השיטה הנוספת לתקשר בין שני התהלכים זה פייפ (pipe) שזהו צינור שמעביר אינפורמציה והוא מחבור לprocesses בעצמם, הצינור יכול להיות חד כיווני או דו כיווני.

בתמונה הבאה ניתן לראות את שלושת השיטות :



הרצאה 5 :

תזמונים - scheduler :

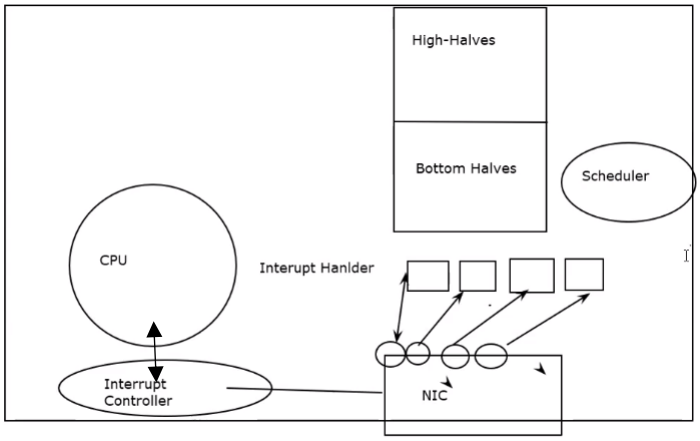
תמונה כללית של המערכת:

התמונה כוללת CPU , תור של חבילות , bottom – halves , high-halves .

ה scheduler מתזמן בעיקר את הbottom -halves .

Interrupt controller– רכיב חומרה.

Interrupt handler – פונקציה.



אנו נתעמק ב scheduler שהוא מתזמן לעבדות (tasks) שנמצאות ב bottom halves שמכיל את הthreads מהkernel או מהprocesses שנמצאים בuser place .

אחת המטרות של הscheduler הוא לנצל את הCPU בצורה כמה שיותר מיטבית ע"י מתן העדפה על בסיס אלגוריתם מסוים.

עבודה מסוג real time מה שהכי חשוב לה זה דד ליין מבחינת זמן ולכן את הscheduler זה מעניין.

ישנם עוד סוגי עבודות שלא דורשות CPU גבוהה.

העבודות שמגיעות לscheduler הם אלו שנמצאות ב state ready לאחר יצירת הprocess.

Scheduler יכול להגיד ל thread שהגיע זמנו להתחלף והthread עושה context switch והscheduler אומר ל thread אחר לתפוס את הCPU.

סוגי scheduler (אלגוריתמים) :

first come first served- **FCFS** – הראשון שבא הוא הראשון שיקבל שירות (fifo), הוא לא כול להעיף thread שלא סיים את זמנו, מתאים לbatch system , לא אפקטיבי ל I/O .

אלגוריתם קצת בעייתי כיוון שהעבודה הראשונה שנכנסת יכולה להיות ארוכה ותוקעת הכל, היתרון שלו הוא שקל לממש אותו.

shortest job first-**SJF** - זמן הסיום הראשון (הקצר) ייכנס לscheduler הראשון, כלומר מהעבודה הקצרה לעבודה הארוכה ,יעיל יותר , מתאים גם ל I/O . מתאים יותר לעבודות קצרות.

אחד החסרונות הוא שקשה להשיג את הזמן הסיום הקצר ביותר , ניתן להשיג אותו מחישובי עבר אך לא תמיד זה זמין.

shortest remaining time first- **SRTF** - וארציה דומה של האלגוריתם הקודם.

highest response ratio next – **HRRN** – נחשב את העדיפות לפי הזמן שבו חיכית כלומר מי שחיכה הכי הרבה מקבל את העדיפות הכי גבוהה.

אלגוריתם שנועד לתקן את האלגוריתם הראשון שבו במקרה והעבודה הראשונה הייתה ארוכה מהרגיל.

Round robin – נותן גדלים (quantum) לכל תהליך ולכל עבודה שקיימת, כלומר כל אחד מקבל זמן מוגבל של CPU ושנגמר הזמן הוא עובר לתהליך הבא וכך חוזר חלילה בלולאה עד שכל התהליכים מסתיימים.

אם ניתן זמן (גודל) יותר מידי גדול (יותר מהזמן של העבודה עצמה) זה יהפוך להיות fifo (האלגוריתם הראשון) כי הthreads יספיקו להסתיים.

ואם ניתן גדלים קטנים הCPU יבזבז זמן גדול יותר על החלפה של הthreads (context switch).

לרוב למערכת הפעלה יש scheduler אחד אך ניתן להחליף אותו.

RT-OS - Real – time operation system - עבודות (tasks) המוגבלות בזמן (דד-ליין) שאנו חייבים לקבל את התוצאה לפני גמר הזמן.

מחולק לשלושה קטגוריות :

safety critical system – אם התוצאה לא תגיע בזמן ההשלכות יהיה הרסניות (דברים רפואיים).

Hard RT system - אמורה לחייב עמידה בזמנים, אין ערך לתוצאה אם היא לא הגיע בזמן (כגון Aתהליך ליירוט טיל או נהיגה אוטונומית).

Soft RT system – רוב הזמן עובדת בדד ליין ועדיין יש ערך לתוצאה למרות שהיא לא הגיע בזמן. (שעון דיגיטלי, נפוץ בעיקר בתחום התקשורת והחבילות).

ישנם המון סוגי מערכות הפעלה המשתמשות בRT כגון blackberry .

הרצאה 6 :

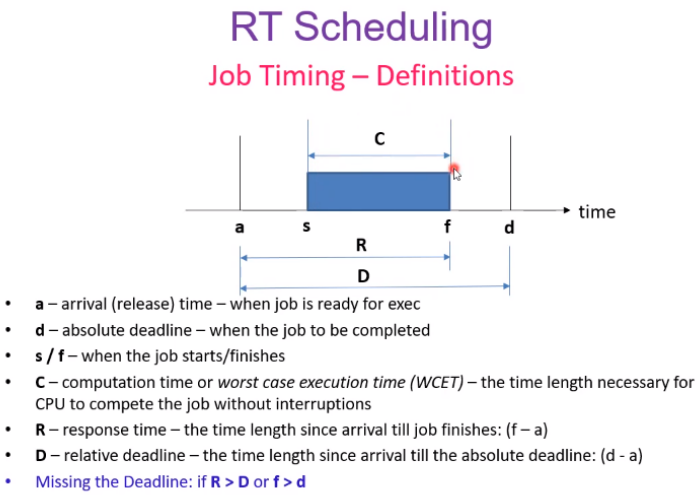
Real -time :

מימוש – ימומש בדרך כלל ב preemtive kernel שהוא בניגוד ל non-preemtive יכול להעיף thread שלא סיים את זמנו (scheduler יכול להעיף הכוונה).

הscheduler מבוסס על עדיפויות שנקבע ע"י מספר אלגוריתמים העיקר שיעמוד בלוח הזמנים.

אין שום עיכובים כלומר הזמן מהלחיצה עד שקרתה הפעולה בפועל ללא עיכובים ובנוסף יש זמן מוגבל לכל משימה מראש.

דוגמא לחישוב אלגוריתם של scheduler (הכוונה לחישוב הזמנים כך שהscheduler יעמוד בדד ליין):



EDF-earliest deadline first – העבודה עם הדד ליין הקצר ביותר מקבל את העדיפות הגבוהה ביותר. בכל נקודת זמן נסתכל למי יש את הדד ליין הכי קרוב ובמקרה שאחד עולה על השני נעיף את ה thread הנמצא בCPU .

במקרה שכל העבודות יעלו מעל אחד כולם יפלו ואף אחד לא יעמוד בדד ליין.

לדוגמא :כאשר T1,T2,T3 הם העבודות כאשר T1 (1,4,4) לדוגמה מקבל בפרמטר הראשון את זמן הCPU בשני את הדד ליין ושלישי את הערוץ מנו הוא נכנס.

- **T1** (1,4,4), **T2** (2,6,6) and **T3** (3,8,8)
- U = 1/4 + 2/6 + 3/8 = 0.250 + 0.333 + 0.375 = **0.958** - feasible

ברור
כי T1 יזכה להיכנס ראשון לCPU כיוון שהוא בעל הדד ליין הנמוך ביותר ובנוסף כל שלושת העבודות יוכלו להתבצע כיוון שסכומם קטן מ 1 (הסכום מחושב כזמן הCPU חלקי הדד ליין).

בדוגמא הבא הסכום גדול מאחד לכן אף עבודה לא תעמוד בדד ליין ויהיה לנו overloading.

- **T1** (2,5,5), **T2** (2,6,6), **T3** (2,7,7) and **T4** (2,8,8)
- U = 2/5 + 2/6 + 2/7 + 2/8 = 0.4 + 0.333 + 0.286 + 0.25 = **1.269** – not feasible

Rm-rate – monotonic - מגיע עם עדיפויות קבועות מראש שהיא זמן המחזור הקצר ביותר (זמן עבודה חלקי דד ליין), הוא לא מסתכל על הדד ליין הוא מסכל על הערוץ ממנו הוא נכנס.

מערכת ההפעלה linux VS. RT :

מערכת ההפעלה linux הוא לא real time אלא הוא משתמש באופן דיפולטיבי בround robin אך ניתן להעביר אותו לreal time כיוון שlinux הוא open source וניתן לשנות את המימוש הscheduler שלו.

Synchronization :

סנכרון תהליכים מתייחס לתיאום בו זמני של מספר processes או threads לסיים משימה כדי לכפות סדר ריצה מסוים ולמנוע race .

הכלי הזה מגן ומיועד למשאבים משותפים כגון מדפסות, קבצים או בסיסי נתונים ,קוד.

CS – critical section – זהו אזור בקוד שאסור שייכנסו אליו יותר מ thread אחד (כלי שמגן).

עקרונות לביצוע מוצלח :

Mutual exclusion - שימוש בCS כלומר מצב בו יש לי טרייד אחד בCS .

נימנע מdeadlock freedom – אם thread אחד או יותר שמנסים להיכנס לCS אזי לפחות אחד ייכנס, אין מצב שאף אחד לא נכנס כלומר הCS לא נעול סתם במקרה ואף אחד לא נכנס זה deadlock freedom.

נימנע מstarvation freedom – אם תהליך מסוים רוצה להיכנס אזי מתישהו הוא ייכנס, לדוגמא יש מצב שבו יש שני תהליכים ורק אחד מהם תופס את כל הזמן של הCPU אנו צריכים לדאוג שהתהליך השני לא יגיע למצב של הרעבה (קבלת זמן CPU נמוך) וכן יקבל זמן CPU.

Logic solution – הפתרון לא צריך להיות תלוי ברכיבי חומרה או במהירות המערכת.

הרצאה 7:

פתרונות ליצירת Synchronization :

Strict alternation – רץ על הprocess עד שהוא מסיים אך יכול ליצור deadlock במקרה שיש תהליך אחד. פתרון נאיבי.

peterson's algorithm – Sleep & wakeup – ישנם שני תנאים לכניסה של הprocess או שזה התור שלנו או שהprocess השני לא מעוניין להיכנס , כאשר כל process מציין אם הוא מעוניין להיכנס או לא מצב שפותר את הבעיה של deadlock אבל עדיין יוצר מצב של הרעבה.

Multi CPU system :

Cache-coherent system – באותו הרגע שה turn השתנה ועד שהוא לא הסתנכרן עם ה cache של CPU מספר 2 אזי כל הגישה ל turn מתעכב אבל כל הסנכרון מתבצע בצורה אוטומטית (קוהרנטי) כלומר שני הCPU רואים אותו דבר (את אותו התור).

non Cache-coherent system – נצטרך לבצע סנכרון בעצמנו (לא קוהרנטי) אחרי כל שינוי בין שני הprocesses.

Test and set lock:

ישנם רכיבי חומרה שכן יכולים לבצע את הסנכרון בין ה caches אחד מהם הוא ה test and set.

לאחר שהערך נכנס (הערך כבר מסונכרן (הכי מעודכן מכל הCPU)) אנחנו נועלים אותו (אף thread לא יוכל לגשת אליו), משנים את ערכו ומחזירים את הערך החדש.

צורת המימוש שלו :

Test-and-set(value)

do atomically
prev:=value
value:=1
return prev

init: value:=0

For each thread, do:

1. ***await*** test-and-set(value) = 0
2. Critical Section
3. v:=0

כלומר עד שthread לא יצא מCS וה value יחזור להיות 0 אזי הthread הבא לא ייכנס ל test and set וכך תהליך זה מתבצע בלולאה עבור כל הthreads .

כלומר אני נועל את הCS ובמקביל כל הטריידים מנסים להיכנס בניגוד לmutex שבו כל הטריידים ישנים במקביל לכן TSL יהיה יותר מהיר.

הרצאה 8:

Semaphores :

זה אובייקט אשר ניתן לחשוב עליו כאל סוג של תמרור בו הוא נותן רק ל process אחד לעבור בכל פעם ואחר מכן לפנות את הדרך לprocess הבא.

ה Semaphores נותן גישה לprocess למשאבים שכרגע רצים כלומר כל process שהSemaphores נתן לו אישור לעבור מקבל גישה למשאבים ורץ.

סוגי Semaphores :

Binary Semaphores – מחולק לשתי פעולות up ו down , כניסה מתבצעת ע"י פעולת down כאשר שני הפעולות האלה הם אטומיות (פעולה רציפה שאי אפשר להפסיק אותה באמצע) שאם הcounter הוא 0 אזי תחסום את הprocess מכניסה (נכנסי אותו למצב שינה) וכאשר הוא 1 נאפס את הtread וניכנס ל CS כאשר ה Up ישחרר את המשאב ל thread הבא.

ישנו מצב בו thread מבחוץ יכול לעשות UP בכוונה ולהיכנס לCS ולקחת משאבים כלומר נוצר מצב בו יש שני threads בCS שזה מפר את עיקרון הExclusion Mutual (בו בCS יימצא רק thread אחד).

הפתרון למצב כזה הוא ליצור מנגנון שרק thread שנגעל יכול לשחרר את המשאבים (הוא הבעלים של המשאב) זה אחת התכונות של mutex וקוראים לה thread ownership .

Counting Semaphores – כאן בניגוד לקודם הcounter יכול להיות מאותחל לN (כלומר N שווה למספר התהליכים שיכולים להיות בCS במקביל) כאשר אם הוא מאותחל ל0 נחסום את ה Process ואם לא נעשה -- counter (בניגוד לבינארי שם נאתחל ל0 בכדי שייכנס).

Mutex - גם כן אובייקט לניהול threads אך עם תכונות שונות :

יש לו בעלות על חוט, כניסה מחודשת לאותו thread.

הבדלים בין binary semaphores ל mutex :

1. ל mutex יש thread ownership – רק חוט שנגעל יכול לשחרר משאבים.
2. mutex תומך ב re-entrance - כניסה מחדש לאותו thread .
3. mutex מממש רק thread שנכנס יכול להשתחרר.
4. mutex תומך בPriority Inheritance ומונע Priority Inversion.

Semaphore semantics :

Num of initial credits +num of up()s = num of threads without going to sleep.

בעיית Producer Consumer :

הגדרת הבעיה : נניח שיש N ליבות CPU ואנחנו מקבלים המון עבודות בלתי תלויות כאשר אנחנו צריכים לחשוב על דרך ליצור מקביליות לעבודות.

איזה מודלים אנחנו יכולים ליצור כדי לקבל מקביליות של עיבוד בצורה הכי אפקטיבית ?

ת. נוכל ליצור N threads או אפילו קצת יותר מN כאשר אנחנו תלויים בעבודות , אם העבדות I/O bound או bound CPU .

במקרה וזה חצי I/O נצטרך פי 2 treads כלומר N2 כי הוא לוקח פי 2 זמן CPU.

פתרון זה מוגבל כי הוא תלוי ב threads ובמקרה שיהיה מיליון עבודות אז אצטרך ליצור לפחות מיליון threads אזי הקצאת הזיכרון, זמן הCPU ושאר משאבי החומרה לא יעבדו בצורה יעילה.

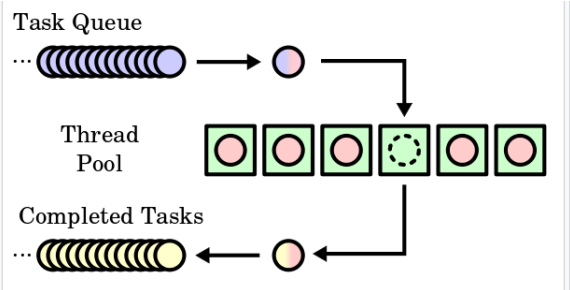
לפתרון היעיל שיוצר הגבלה קוראים **thread pool -** ה treads האלה נקראים costumers שמוצאים את העבודות ומתחילים לעבד אותם מול איזו בריכת חוטים קטנה שנקראים producers שתפקידם לייצר את העבודה (למשל מקבילים חבילות מהרשת בודקים אותם ומכניסים לתור מסוים) כאשר יש הגבלה לתור, כלומר N מוגבל כדי שלא יוצר מצב בו יש יותר עבודות מאשר לcostumers לעבד, בנוסף יש כאן mutex לשימוש בתור כאשר הוא 1 או 0 .

בנוסף הפתרון מורכב גם משני אובייקטים מסוג semaphore empty/full שתפקידם לספור כמה משימות נכנסו וכמה יש בתור בכדי לא לחרוג מN .

Question 2 : Producer-Consumer Problem

```
#define      N       100          /* Buffer size */
Mutex        UseQ = 1;            /* access control to CS */
semaphore    empty = N;           /* counts empty buffer slots */
semaphore    full = 0;            /* counts full buffer slots */

void producer(void){
   int item;
   while(1){
        produce_item(&item);      /* generate something... */
        down(&empty);             /* decrement count of empty */
        down(&UseQ);              /* enter critical section */
        enter_item(item);         /* insert into buffer */
        up(&UseQ);                /* leave critical section */
        up(&full);                /* increment count of full slots */
   }
}
```

סכימה להמחשה :



A sample thread pool (green boxes) with waiting tasks (blue) and

הרצאה 9:

Deadlock :

קיפאון –s מצב בו שתי פעולות (threads) מתחרות מחכות כל אחת לסיומה של האחרת, ומכיוון שכך, אף אחת מהן אינה מסתיימת כלומר, שתי threads ממתנים שניהם לתגובתו של השני ניתן לחשוב על זה בצורה של שני אנשים עומדים מול דלת פתוחה וכל אחד מחכה שהשני ייכנס וכך נוצר מצב ששניהם לא נכנסים.

4 תנאים ליצירת deadlock :

1.mutual exclusion - מניעה הדדית – משאב נשלט רק ע"י process או thread אחד.

2.hold and wait - החזק והמתן – אחד התהלכים תופס משאב ומתחיל לחכות לעוד משאב (תופס semaphore ומחכה לעוד semaphore)

3. no preemption – אין הפקעה – אף אחד לא יכול להעיף אותך מהמשאב שתפסת עד אשר לא תשחרר אותו.

4.circular wait – המתנה מעגלית – שתי תהליכים או יותר מחכים למשאבים שנתפסים ע"י תהליכים אחרים כלומר קבוצת התהליכים יוצרת מעגל בו כל תהליך מחכה למשאב המוחזק על ידי תהליך אחר. רק את התנאי הזה אנחנו יכולים למנוע.

מניעת deadlock :

א. ניצור מצב בו אחד מארבעת התנאים אינו מתקיים וכאמור זה יכול להיות רק התנאי הרביעי.

ב. נקצה כמות משאבים גדולה כך שלא יוצר מצב בו "חסר" משאב.

ג. ניצור מעגל של תהליכים מול משאבים שבו המשאבים תמיד יהיו פנויים ביחס לתהליכים ניצור מצב כזה ע"י זה שנהרוג תהליכים .

The banker's algorithm :

אחד מאלגוריתמים למניעת deadlock.

בעל כמה מצבים :

1. Safe - כל המערכת רצה ומשתמשת בכל השירותים ולא נכנסת לdeadlock.

2. Unsafe - חלק מהמערכת תרוץ וחלק תיכנס לdeadlock.

3. Deadlock – נכנסו ל deadlock ותקועים.

האלגוריתם בנוי מווקטורים ומטריצות.

וקטורים :

E – מספר המשאבים הקיימים מכל סוג.

P – מספר המשאבים מכל סוג הנמצאים בתהליך עיבוד.

A – מספר המשאבים הזמינים מכל סוג.

מטריצות :

C – מטריצת הקצאת הנכנסים הנוכחית (אלוקציה).

R – מטריצת הבקשות.

מהלך האלגוריתם :

1. נסתכל במטריצת הבקשות (R) ונמצא שורה בה כמות המשאבים קטנה או שווה לכמות המשאבים הזמינים מכל סוג (A) אם לא קיימת שורה כזו יכול להיווצר מצב של כניסה לdeadlock .

2. לאחר מכן נניח שכאשר העיבוד של השורה אותה בחרנו מסתיים הוא משחרר את כל המשאבים לווקטור A ונסמן אותו.

3. נעבור שוב על פעולות 1 ו 2 ונראה כי כל השורות מסומנות וכל התהליכים מסתיימים.

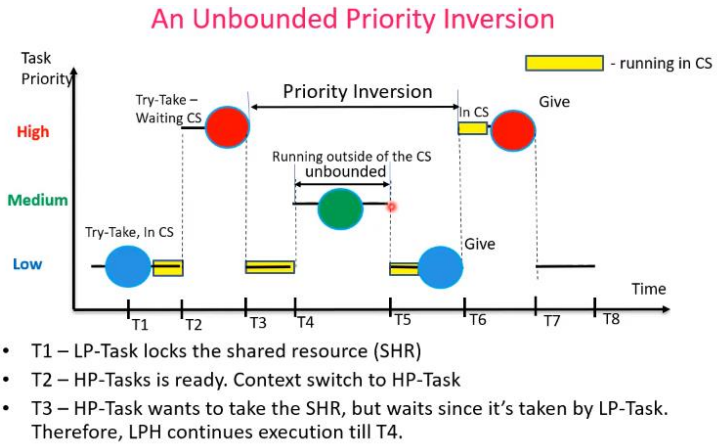
אם הם מסתיימים אפשר להגיד שהיינו במצב safe ואם לא כולם מסתיימים אזי ניתן להגיד כי קרה deadlock ואנחנו במצב unsafe .

<u>Priority inversion - היפוך עדיפויות:</u>

היפוך עדיפות הוא תרחיש בתזמון בו משימה בעדיפות גבוהה מונעת בעקיפין על ידי משימה בעדיפות נמוכה יותר ובכך הופכת את סדרי העדיפויות היחסיים של שתי המשימות.

זהו אחד ההבדלים בין mutex ל semaphore (יש ל mutex אך אין לsemaphore).

לדוגמא unbounded :



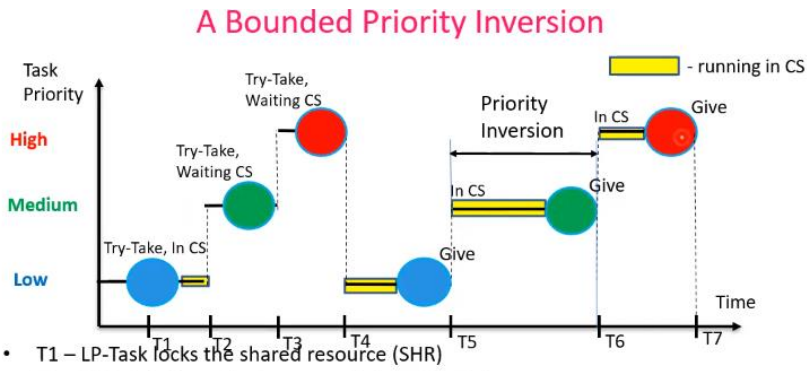
ניתן לראות בתמונה כי העבודה עם העדיפות האמצעית (בצבע ירוק) מקבלת זמן CPU בעוד העבודה בעלת העדיפות הגבוהה (בצבע אדום) לא מקבלת זמן CPU כיוון שהעבודה בעלת העדיפות הנמוכה (בצבע כחול) תפסה את CS ואילו להמתנה הגיעה העבודה בעלת הבצע האדום והתהליך הירוק הצליח להשתחל ולהיכנס לCPU.

<u>איך ניתן למנוע היפוך עדיפויות?</u>

כיוון שב mutex יש את התכונה של thread ownership אנחנו יכולים לדעת מה id של הthread שתפס את CS ואז ניתן לו העדפה גבוהה כלומר יצרנו מצב בו עכשיו הכחול שתפס את הCS שווה לאדום שממתין בתור ובטוח שהתהליך הירוק לא ייכנס ויקבל CPU.

פתרון זה לא מונע deadlock.

דוגמא ל bounded :



- T1 – LP-Task locks the shared resource (SHR)
- T2 – MP-Tasks is ready. Context switch to MP-Task
- T3 – MP-Task wants to take the SHR, but waits since it's taken by LP-Task.
- T3 – HP-Task is ready. Context switch to HP-task
- T4 - HP-Task wants to take the SHR, but waits since it's taken by LP-Task.

בתמונה זו ניתן לראות כי התהליך הכחול בעל העדיפות הנמוכה שוב תופס את CS אך הפעם ממתין בתור התהליך בעל העדיפות הבינונית (בצבע ירוק) ורק אז התהליך האדום בעל העדיפות הגבוהה.

הגורם לתופעה הזו היא שימוש בתור רגיל ולא בתור עדיפויות.

הרצאה 10 :

CPP – ceiling priority protocol :

פרוטוקול זה גם מונע deadlock וגם היפוך עדיפויות והוא מתאים גם ל mutex וגם לsemaphore .

מושגים :

Ceil(i) – הערך הגבוהה ביותר של semaphore כלומר i מייצג את עדיפות ceil יחזיר את העדיפות הגבוהה ביותר .

Task (i) – יש כאן הסתכלות על כל מכלול הsemaphores ועל כל מכלול העבודות לפני שניתן לעבודה מסוימת לנעול את הsemaphore כלומר עבודה תינעל אם התיעדוף שלה (i) גבוהה ממש מהתקרה של כל התהליכים שכרגע נעולים ע"י משימות אחרות.

תנאי 1 – אם task (i) לא מתקיים כלומר אם גילינו שלא ניתן לעבודה לנעול semaphore יכול להיות שישלחו אותו לתור ל semaphore אחר.

תנאי 2 – אם תנאי 1 בוצע כלומר task(i) ננעל ע"י *S אז עדיין עושים השוואת עדיפויות (עושים Priority Boosting).

התהליך עם התיעדוף הגבוהה ביותר יקבל את המאשב הגובהה ביותר.

תכונות :

OCPP – כאשר עבודה אחרת מנסה לגנוב משאב של עבודה שנעולה עליו אזי קורה הboos לתקרה של הsemaphore.

ICPP – ה boosting (השוואת העדיפויות)מתבצע אוטומטית ברגע שהוא נכנס לתור כלומר ברגע שהוא מקבל משאב.

בעיית הפילוסופים:

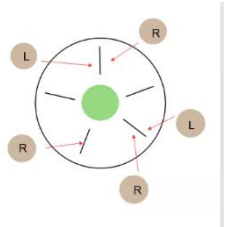
תיאור הבעיה : ישנם חמישה פילוסופים שרוצים לאכול וכדי לאכול הם צריכים להשיג שתי מזלגות אחת מכל צד אך יש רק חמישה סה"כ.

אז בהתחלה הם מנסים לתפוס רק את מה שמימין ולכן אף אחד לא יוכל לאכול.

מצב זה מתאר deadlock ודאי (wait and hold).

פתרון הבעיה : textbook – נחלק לכל אחד צלחת משלו (semaphore)ובמקום שכל אחד ינסה לאכול מהצלחת מהמרכזית (שהיא תיהיה הmutex) אזי אם תפסת שני מזלגות תאכל ותמשיך , לאחר מכן ננסה לבדוק אם אחרי האכילה של מי שתפס אם ניתן יהיה להכניס את אחד החברים לאכילה ע"י ניתנת המזלגות וכך אין אחד שמגיע למצב של הרעבה וכולם אכלו במקסימום תמתין סיבוב אחד או שתיים עד שתאכל.

LR solution – נחלק את הפילוסים לשני קבוצות ,קבוצה ראשונה תיקח מימין ואילו השניה תיקח משמאל ואז יוצא מצב שכל זוג נאבק על מזלג אחד.



כאשר כל הפילוסופים לוקחים ומשחררים את המקלות.

גישה זו מנסה למנוע deadlock אך גם מונעת הרעבה.

טבלת סיכום לSynchronization:

שם	תיאור
Strict alternation	
peterson's algorithm	
Cache-coherent system	
TSL	
Semaphores binary/counter	
Mutex	
The banker's algorithm	
CPP – ceiling priority protocol	

זיכרון – Memory :

פעם כאשר מתכנת היה כותב תוכנה הוא היה מתאים אותה לכתובות הפיזיות בזיכרון וכאשר הוא היה רוצה להעביר את הקוד למחשב אחר נוצרה בעיה כיוון שבמחשב השני הכתובות הם אחרות.

כלומר פעם כל תוכנה שהיית כותב הייתה תלויה ברכיבי החומרה של אותו המחשב עליו הקוד נכתב בצורה ספציפית.

כדי לפתור את הבעיה הזו הומצא מנגנון בשם **HW (hardware) independent memory addressing** שתפקידו להמיר את הכתובות הפיזיות לכתובות זיכרון וירטואליות כאשר המתכנת יעבוד מול הזיכרון הווירטואלי ללא תלות בחומרה.

הזיכרון הווירטואלי מורכב מ Virtual address = Physical Address + Normalization Offset כאשר הוא מתחיל מכתובת ידועה.

לאחר שחילקו ל kernel/user space וmultiprogramming נוצרה בעיה נוספת כי עכשיו היו צריכים להקצות יותר זיכרון להרבה יותר משאבים במקביל ועכשיו גם הזיכרון מחולק ולא אחיד.

כלומר יש לנו זיכרון מוגבל שמספר תהליכים רוצים לנצל את כולו והמחיצות בינהם הגבילו אותנו בכמות הזיכרון הקבוע שניתן לחלק.

אזי בהתחלה כל תהליך היה מקבל חתיכה קבועה מהזיכרון למשל, אם יש שלושה תהליכים אז היו מחלקים את הזיכרון לשלוש וכל תהליך היה מקבל חלק.

פתרון זה היה לא היה יעיל כיוון שהתהליך לא היה מנצל את כל הזיכרון ובנוסף גם לא היה אפשר לנצל את הזיכרון בצורה דינאמית.

אזי יצרו את רכיב הזיכרון הקטן שנקרא page שמתי שאתה צריך אותו אתה מקבל אותו, הוא קטן ודינאמי ולכן איפה שנדרש אתה מקבל אותו ניתן לחשוב על זה כמו טפטפת לצמח במידה הדרושה עם הכי הרבה חסכון בזיכרון.

כלומר מתי שאתה צריך ה Process יקבל page זיכרון הפיזי שיתמפה לתחום כתובות הווירטואלי וייתן לו כיסוי לעבודה.

כעת הCPU צריך כתובת פיזית כדי לעבוד איתה (כתובת בRAM עצמו) ולכן נצטרך להמיר מהכתובת הווירטואלית לפיזית זה נקרא **page table.**

ישנו גם מנגנון בשם **memory mapping -** מנגנון שנותן לכל process יחידות זיכרון קבועות שלא מחולקות בחוצצים כמו המודל הקודם אלא מוקצות בצורה יותר דינאמית ובמקטעים שלמים. כל אחד מהם יקבל Space Memory וכך נוכל לתמרן ביניהם בצורה הרבה יותר נוחה. בשורה התחתונה, כל תהליך היה מקבל תחום כתובות וירטואליות מלא משל עצמו ויוצר גמישות גדולה יותר בין זיכרון של הprocess .

גודל של page - נע בין 1/4/8/16 kb , יחידת זיכרון קטנה שתפקידה זה להיות הבסיס הקטן לזיכרון וירטואלי.

הרצאה 11 :

חזרה וחידוד על זיכרון כמו שניתן לראות בסוף הרצאה 10 שם המושג הורחב.

Page table :

גודל הטבלה יכול להיות בזיכרון לוגי של 32 ביט בגודל 4 GB והוא יתחלק בצורה הבאה (בהתאם לגודל הpage) :

א. 4 מיליון pages בגודל של kb1 .

ב. 1 מיליון pages בגודל של kb4.

חישוב גודל הטבלה :

Page 16K - 4 bytes/entry x 256 K entries = 1 Mb
Page 4K - 4 bytes/entry x 1M entries = 4 Mb
Page 1K 4 bytes/entry x 4M entries = 16 Mb

כאשר כל כניסה בבסיסה שוקלת כ4 בתים ונכפיל בכמות הדפים וזה ייתן לנו את תפיסת הזיכרון שנלקחת.

מסקנה: אנו רואים שככל שה Page-גדול יותר, כך הטבלה הסופית (המערך) שאנו צריכים לזיכרון הוא קטן יותר ולהיפך. המצב הזה חוסך לנו מקום ומשאבים לגודל הטבלה אבל זה ההפסד מזה שהוא שהדפים יכולים להיות לא מנוצלים מספיק בגלל הגודל שלהם.

אם הדף קטן יותר אני ארוץ לאט יותר אך ניצול הזיכרון יהיה מיטבי.

Two level page table :

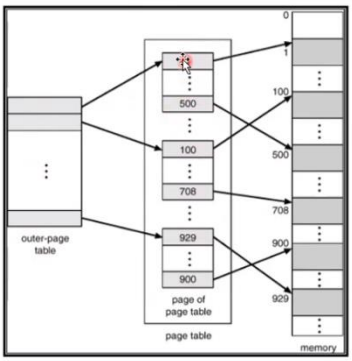
מטרת הטבלה בעלת שתי השלבים היא להחזיק טבלאות גדולות בחומרה ,זה לא אפשרי אבל ניתן להחזיק אותם בזיכרון הראשי (main memory).

במקום להחזיר איזה מערך רציף שמכיל את כל הטבלאות ניתן לארגן טבלה שמצביעה לטבלה.

החיסרון נובע מהטבלה החיצונית בה לא בטוח שהיא תאוכלס במלואה.

דרך זו חוסכת בזיכרון על הטבלה עצמה.

Two-Level Page-Table Scheme



לדוגמא : יש כתובת וירטואלית על 32 bit שכוללת 32 בתים כאשר גודל הדף הוא 4 k (שמייצג כתובות מ 0-4095) שזה שווה ל 2^12 כלומר 12 ביטים (כאשר 1k זה 2^10).

הכתובת מחולקת ל page number ו page offset כאשר ה page offset כאמור יהיה בעל 12 ביטים.

נשארו לנו 20 ביטים שאותם נחלק ל page number :

page number		page offset
p_1	p_2	d
10	10	12

כאשר p1 יהיה עבור הטבלה החיצונית ואילו ה p2 עבור הטבלה הפנימית.

ש. כמה זיכרון תתפוס הטבלה כדי שהיא תוכל לתמוך בנתונים הבאים :

4mb of stack

4 mb of core segment

4 mb of heap

ת. עובר כל אחד ממבני הנתונים האלו אנו יודעים כי עבור 4mb נדרש ל 1k של דפים (ראה חישוב טבלה לעיל) .

כעת את ה 1k דפים נכפיל ב 4k בתים שזה 4k בתים שזהו דף אחד.

כעת יש לנו שלושה דפים.

כעת ניתן להניח שהזיכרון של ה heap וה stack רצוף וכיוון שהוא רצוף הוא לוקח דף אחד אבל הזיכרון של הcore segment יכול לקחת 4 דפים אזי נקבל 6 דפים.

כאשר כל דף הוא 4 kb אזי נקבל 24 kb וזה יהיה הגודל של הpage table .

Inverted page tables :

יכול להיווצר מצב בו הטבלה שוקלת יותר מדי לדוגמא במערכת של 64-bit כאשר גודל דף הוא 4k כפול 2^52 כפול 8 בתים (משקל של כניסה למערכת של 64 בתים) זה יכול להגיע ל30M לטבלה!

הפתרון היה למפות בצורה הפוכה כלומר, במקום למפות פיזיות לכתובות וירטואליות נמפה כתובות וירטואליות לכתובות פיזיות.

כעת נשתמש בזיכרון הRAM ולכן גודל הטבלה יהיה 2MB.

כלומר ניקח את הRAM ששקול 1 GB (כלומר מיליארד ביטים)ולצורך העניין נחלק אותו ב4k שהוא גודל הדף ונקבל 256 כניסות (כלומר נצטרך 256 שורות) כעת נכפיל ב8 בתים עבור על כניסה ונקבל 2mb .

כדי להגיע מכתובת וירטואלית לכתובת הפיזית נארגן את הcache בhashTable.

כעת ניגש לכל כתובת של תהליך (PID) שהוא ייחודי וכן נקבל כתובת פיזית שונה.

PTE – page table entries :

הניסה מורכבת מביטים המייצגים את הכתובת הפיזית.

כאשר יש כמה תנאים שכל תנאי מייצג ביט.

האם היא בזיכרון ? במידה וכן שווה לביט 1 .

האם היא בשימוש או לא ? במידה וכן שווה לביט 1.

יש זכיות גישה וזה לא רק דפים לקריאה בלבד (כמו הROM) גם כן משפיע על כמות הזיכרון.

TLB – translation lookaside buffer :

באפר צדדי.

בהתחלה הCPU לא היה מתוחכם ועשו הכל בתוכנה.

אזי יצרני החומרה יצרו MMU (יחידת ניהול זיכרון) ומשם התחכום לCPU התחיל.

כאשר בMMU הכניסו את הTLB כלומר ההתאמה בין הכתובת הפיזית לווירטואלית הפכה פשוטה יותר.

ש. מה ההבדל בין context switch של threads של אותו process לבין processes שונים ?

ת. ברגע ש TLB מומש עם כניסה אחת אזי context switch ששייכים לתהליכים שונים במצב הזה מביאים למצב של Flush-TLB והעמסה שלTLB ים חדשים, זה למה במקרה הזה ייקח לנו יותר זמן עיבוד.

יש לציין כי TLB קיים ב CPU וגם קיים בצד התוכנה.

תהליך בקשה – resolving :

א. הprocess/kernel פונה לכתובת הווירטואלית.

ב. הם שואלים את ה MMU שהוא פונה לחומרה ומבקש.

ג. אם הכתובת קיימת בMMU הוא מחזיר את הכתובת ואם לא מטיילים ב page table ומעדכנים עבור הTLB זה במקרה והכתובת קיימת והיא חוקית אבל הMMU לא החזיר אותה.

ד. במקרה והכתובת לא חוקית נקבל שגיאת PAGE_FAULT בעלת שתי סוגית hard ו soft.

Soft– הדף קיים אבל מוקצה לתהליך אחר וניקח דף של תהליך אחר.

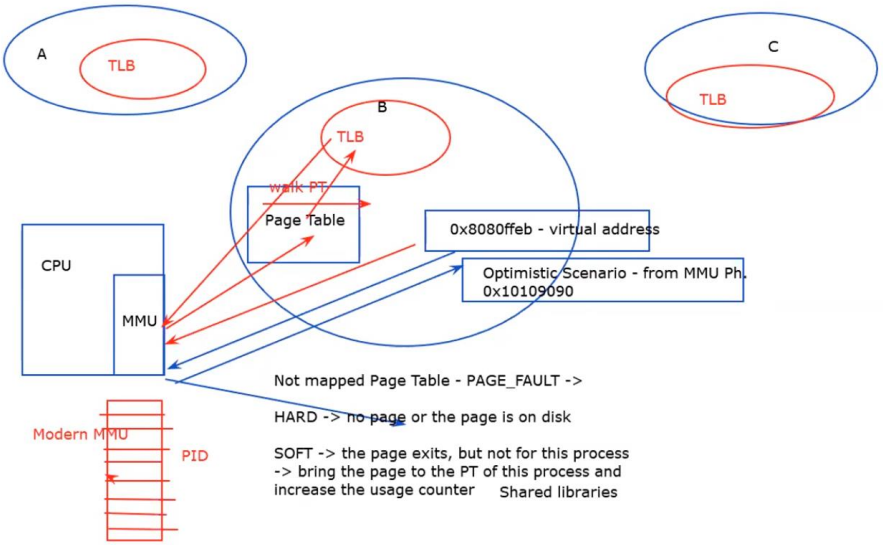
Hard– הדף לא קיים בזיכרון בכלל ולכן נקצה דף חדש.

Hard & soft - למפות את הדף שקיבלנו ולהכניס אותו לpage table.

גם כאן נסרוק את ה page table בצורה ידנית ונעדכן את הTLB .

הרצאה 12 :

סכימה על תהליך הבקשה של TLB :

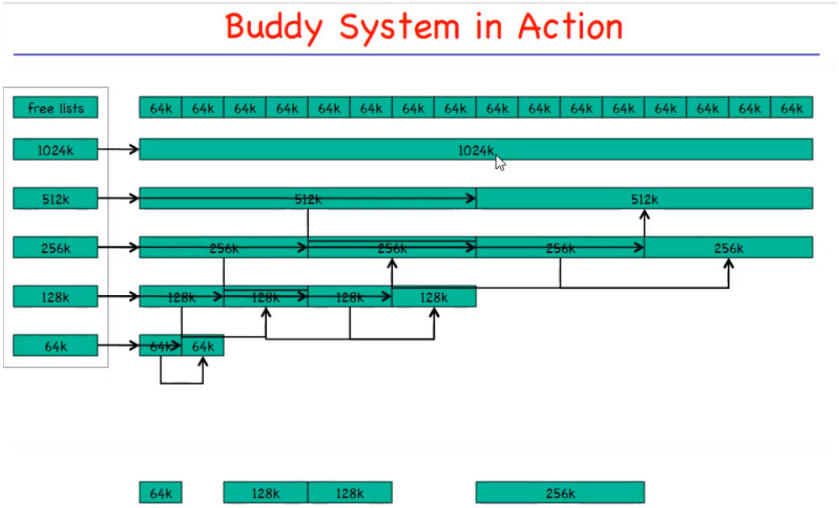


Dynamic allocation - אגירה דינאמית :

נניח שנרצה להגדיר הקצאות של זיכרון בעזרת malloc/calloc , new וכו' אזי ה system call יהיה לספריה GLIBC.

הספריה הזו משתמשת בלוגיקה של knuth's buddy allocator שחותך את הזיכרון ומגדיר את הקצאת הזיכרון ל malloc/calloc וכדומה .

הלוגיקה עובדת בצורה הבאה :



נחלק כל פעם ב2 עד שנגיע לגודל המתאים, נעצור כאשר נגיע לזיכרון המבוקש ואז הוא ייתן לאפליקציה את הזיכרון כאשר כל פעם נעשה merge אחורה כדי להקצאות את הזיכרון.

לדוגמא אם נרצה זיכרון של 30 k אז נתחיל ונחלק עד אשר נגיע לדף של 64 k אותו נחלק ל2 ונקבל 32 כעת נעשה Merge ונחזיר את הk 64.