

Process Management



© 2022 מערך הפעלה

Operating Systems

Lecture 2 – Process Management

Dr. Marina Kogan-Sadetsky

Course Syllabus

- 1. Introduction
- 2. **Process Management**
 - Process states and structures
 - Process management
 - Inter-process communication
 - Signals
 - Threads
 - Specific implementations
- 3. Scheduling algorithms
- 4. Synchronization
- 5. Memory Management
- 6. File Systems
- 7. Virtualization

הערות כלליות ותזכורות:

- א. הפרק הראשון בקורס.
- ב. בהרצאה הקודמת דיברו כי בבלוק משנים סטטוס, מסייםים לגבות ולא נתונים לו זמן מעבד.
- ג. המטרה של מערכת הפעלה היא לנצל את המעבד למקסימום.
- ד. אם מערכת הפעלה מזיהה שמדובר בתוכנית אינטראקטיבית, היא מספקת את הזמן הדרוש לקבלת תגובה מהיר.
- ה. כבר דנו בנושא זה במסגרת הקורס SPL, אך כעת עמוקיק בו.

Process management – main goals

- ❑ Manage **process entity** - create, delete, suspend, block, ...
process table (PCB – process control block, PTE – process table entry) data structure is needed
- ❑ Allocate resources for process
efficient policy is needed
- ❑ Interleave execution of processes to maximize **CPU utilization**
scheduler is needed – maximal utilization of CPU time (also try to maximize other resources utilization)
- ❑ Provide reasonable **response times**
- ❑ Support **inter-process communication** and **synchronization**
process priority is needed – there is a user waiting for (fast) response

מודל התהילך:

התהילך הוא מופע של תוכנית בעת ריצתה, הכוללת ערכיו ה-*program counter*, רגיסטרים ומשתנים. לכל תהליך יש "מעבד" וירטואלי משל עצמו. בפועל המעבד מחליף בין תהליכיים כל הזמן, טכניקה אשר נקראת ריבוי תהליכיים.

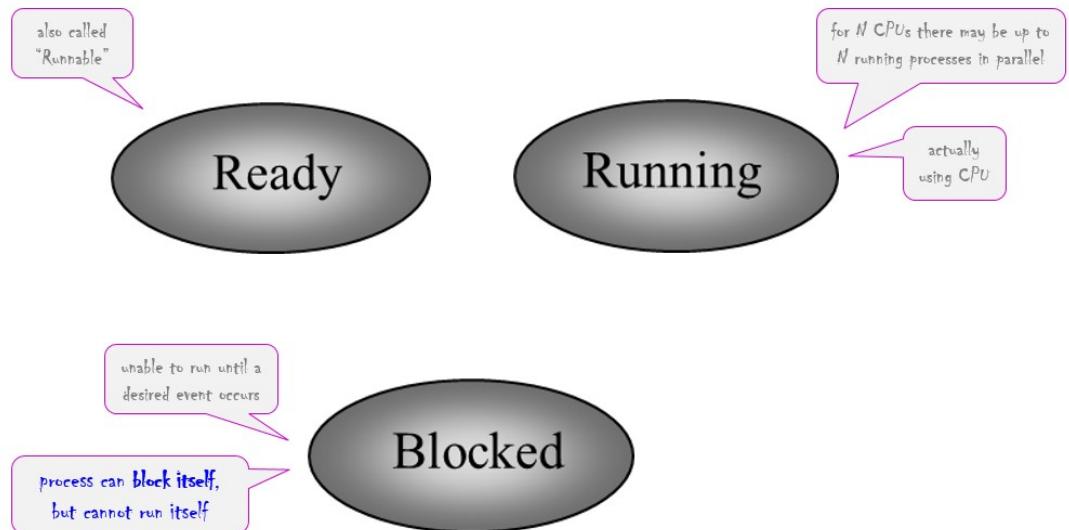
מטרות ראשיות:

- א. ניהול קיומם של תהליכיים.
- ב. הקצאת משאבם ל手続きים.
- ג. למקסם עבודת המעבד, למקסם את השימוש במעבד.
- ד. לספק זמני תגובה היגוניים: אם יש פסקה אז אנחנו רוצים שתהליך יגיב כמה שיותר מהר לזה.

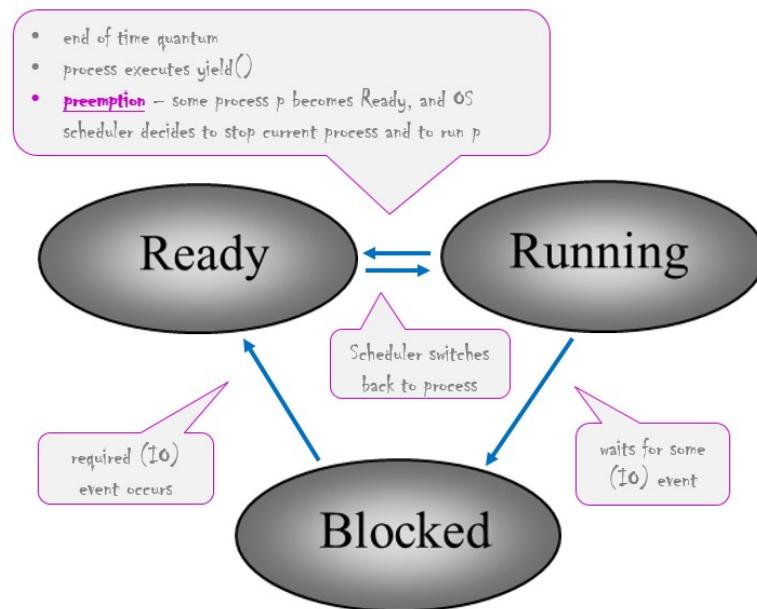
Process Management : outline

- ❑ Process states and structures
- ❑ Process management
- ❑ Inter-process communication
- ❑ Signals
- ❑ Threads
- ❑ Specific implementations

Process Basic States



When do transitions occur?



במקרה הבסיסי תהליך יכול להיות ב-1 מבין שלושת הממצבים הבאים:

Running:

תהליך במצב זה כאשר הוא משתמש במעבד, בambilים אחרים יש לו CPU שמקצה לו. לכן מספר התהליכיים אשר יכולים לרוץ בו בזמןות הוא מספר הליבות.

Blocked:

תהליך נמצא במצב זה כאשר הוא לא יכול לקבל זמן ריצה עד שאירוע חיצוני יתרחש.

Ready:

תהליך במצב זה כאשר הוא יכול לקבל זמן ריצה, אך הוא נוצר לזמן קצר על מנת לאפשר להתהליכים אחרים לרווח. יכול לקבל זמן ריצה.

מעברים בין מצבים:

לפי המודל הבסיסי של 3 מצבים ישנו רק 4 מעברים אפשריים.

מעבר אחד מ-Blocked ל-Running: היה אירע SO, ככלומר התהליך ננע על התקן קלט-פלט.

מעבר מ-Ready ל-Blocked: האירע שבילו תהליכי המתינו התרחש.

מעבר מ-Running ל-Ready: המתזמן החליט להקצות זמן ריצה לתהליכי אחר.

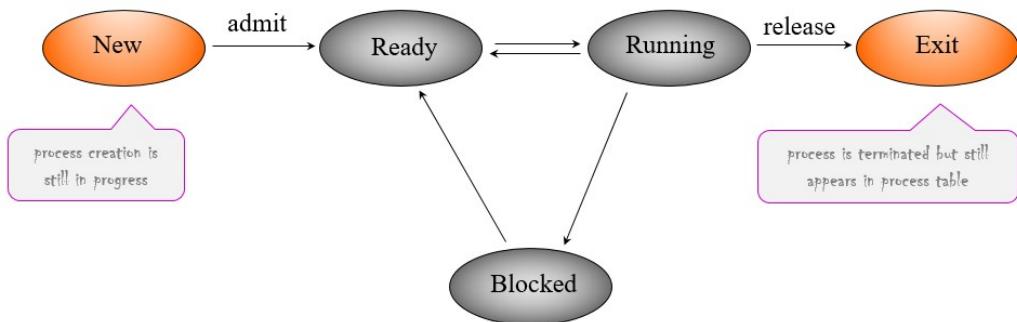
מעבר מ-Running ל-Ready: המתזמן החליט להקצות זמן מעבד לתהליכי.

ההחלפות בין התהליכי תלויה גם בסוג של אלגוריתם התזמון, ככלומר אם הוא preemptive או לא.

אם הוא לא: הוא רץ עד שהוא מסיים או שהוא מחזיר זמן ריצה.

רוב המתזמנים במערכות הפעלה הם preemptive.

Five-State Process Model



מודל חמשת המצבים

ברגע שאנו עוברים לסביבת ריבוי תהליכי, יש לזכור בחשבון מצבים אפשריים נוספים:
כעת יש עוד מצבים:

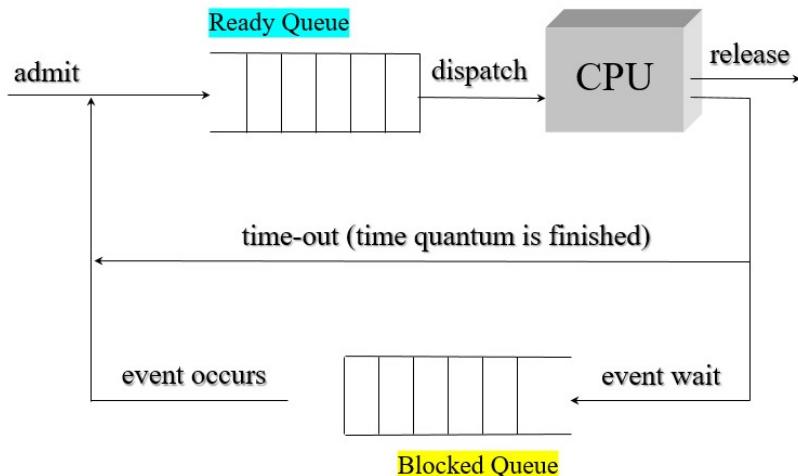
new:

כלומר התהליך עדין לא נוצר. זהו השלב שבו מערכת הפעלה בונה את מבני הנתונים הרלוונטיים לתהליכי. בcontra זו מערכת הפעלה יכולה למנוע מצב בו תהליכי אחד נדרס על ידי תהליכי אחר בעת הייצרוותם.

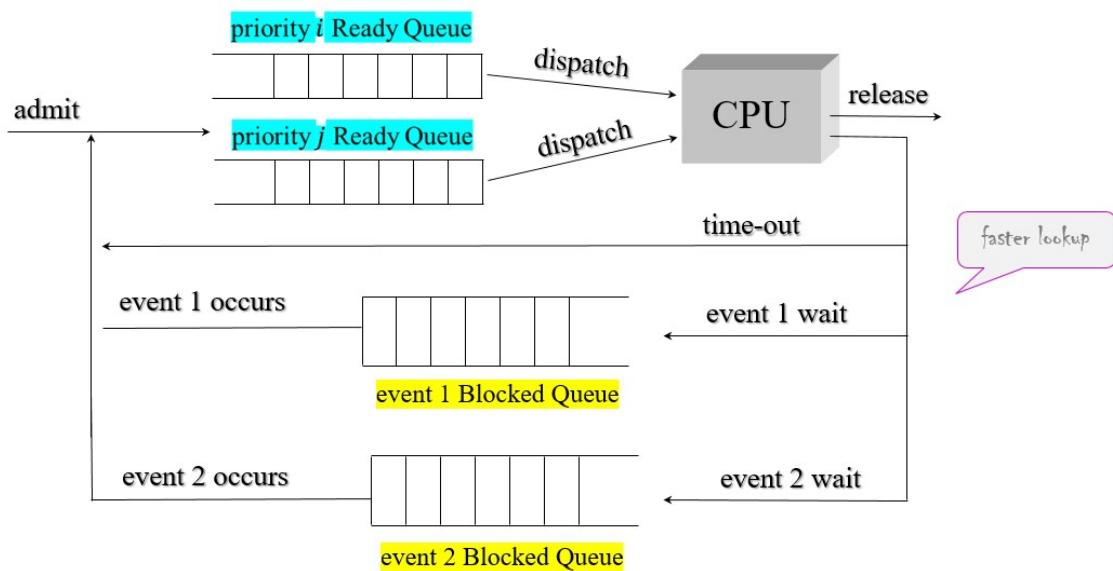
exit:

התהליכי הסתים בעזרת קריית מערכת, או בגלל פעולה לא חוקית שגרמה לשגיאת זמן ריצה.

Scheduling: Single Queue



Scheduling: Multiple Queues



בעת דיבור על הזמן התחליכים, קיימים שני מגבלים עיקריים - זמן עמ תור אחד וזמן עם מספר תורים.

בזמן עם תור אחד, השלבים השונים של התהיליך הם: admit, ניתוב התהיליך ל-Ready Queue, dispatch, CPU, release, time-out אם ה-time quantum נגמר. אם הוא ממතין ל-event מסוים, הוא עובר ל-Blocked Queue עד שה-event יתרחש.

מדובר על שני תורים מרכזיים:

1. Ready Queue: שבו התחליכים מקבלים לקבלת זמן ריצה.
2. Blocked Queue: מסויים ולאחר מכן מקבלים זמן ריצה כרגע event-happened-הכולל את התחליכים שemmattinim ל.

שימו לב שהמידע המתואר בתורים אינו נמצא פיזי בהם, אלא מצביע אליהם באמצעות מצביעים למיקומים בטבלה.

בתמונה עם מספר תורים, כל תחילה עשוי לעבור בין מספר תורים לפי סדר העדיפות שלו. השלבים פה דומים, אך התהליכים מונחים באופן היררכי יותר בהתאם לסוג event שלהם. במקרה זה, קיימת אפשרות שתהליכיים בעלי עדיפות נמוכה לא יקבלו זמן ריצה בכלל.

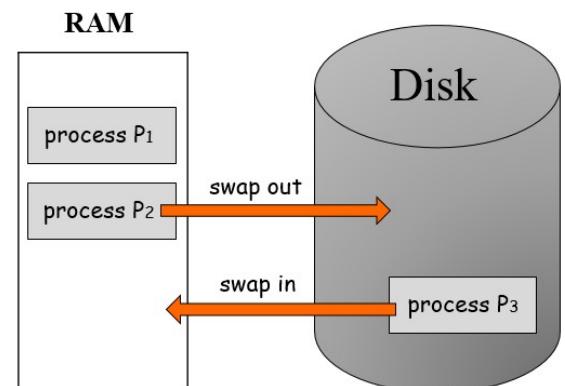
אחת המכלולות החשובות היא **ה-timeout**, שהוא האות שהתהליך מקבל כאשר הזמן המוקצב לו נגמר.

Suspended Processes

- ❑ CPU is much faster than I/O, so many processes could be waiting for I/O

- ❑ **Swap** some of these processes to disk may free up memory

- ❑ Blocked state for swapped process is called **Blocked-Suspended**, Ready state is called **Ready-Suspended**



Swapping - Unix example

❑ When is swapping done?

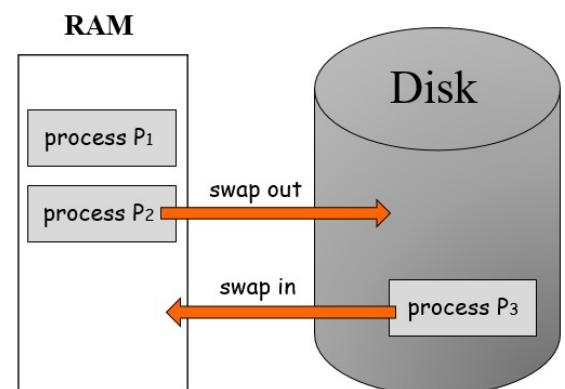
- kernel runs out of memory
- fork() has no space for child process
- brk() has no space to expand Heap segment
- kernel has no space to expand Stack segment
- READY swapped out process with high priority

Is swap-out expensive ?

No. Most of process image is already on Disk, so we need to write to Disk only the modified data.

❑ Who is swapped?

- swapped-out READY processes with high priority
- swapped-out large processes with low-priority



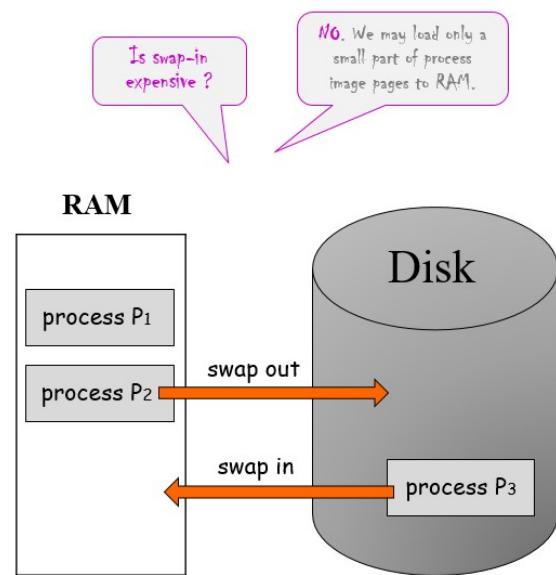
Swapping - Unix example

❑ When is swapping done?

- kernel runs out of memory
- fork() has no space for child process
- brk() has no space to expand Heap segment
- kernel has no space to expand Stack segment
- READY swapped out process with high priority

❑ Who is swapped?

- swapped-out READY processes with high priority
- swapped-out large processes with low-priority



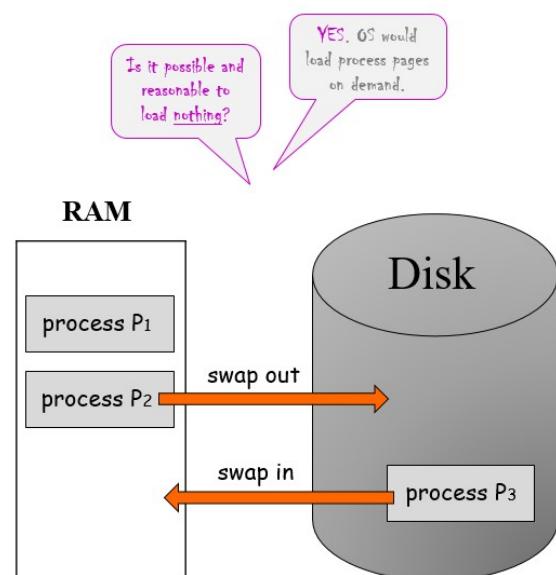
Swapping - Unix example

❑ When is swapping done?

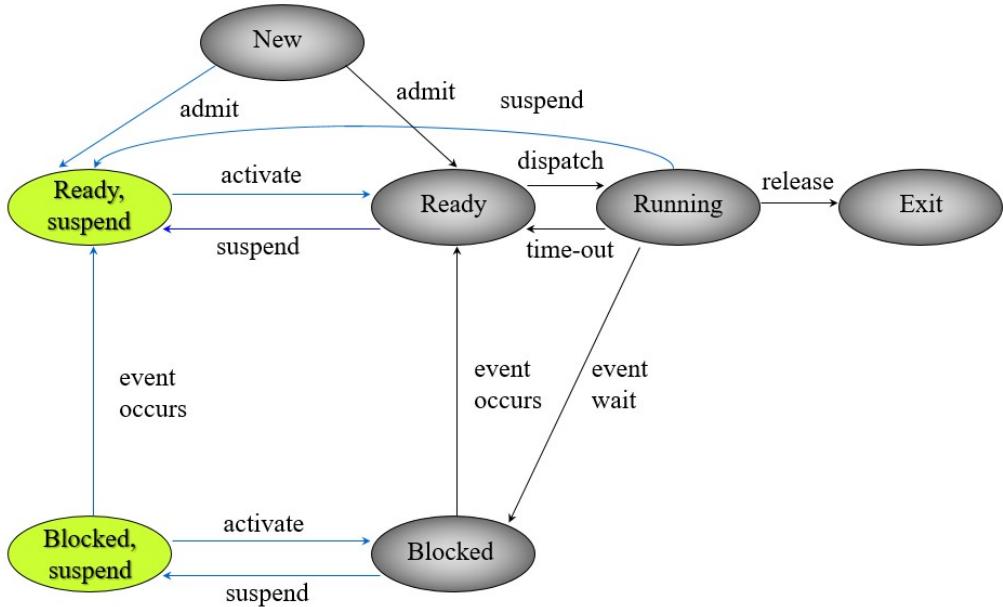
- kernel runs out of memory
- fork() has no space for child process
- brk() has no space to expand Heap segment
- kernel has no space to expand Stack segment
- READY swapped out process with high priority

❑ Who is swapped?

- swapped-out READY processes with high priority
- swapped-out large processes with low-priority



Process State Diagram with Suspend States



שעיות תהליכיים

הבעיה:

המעבד מהיר הרבה יותר מהתकני קלט-פלט. זה אומר שיתיכן שרבים מהתהליכיים יהיו במצב המתנה להתקנים אלו.

פתרונות:

במהלך זמן המתנה, אפשר להעביר את התהיליך מהזיכרון אל הדיסק. כאשר התשובה מההתקן מתقبلת, התהיליך מוחזר לזכרון. הפעולה זו נקראת "החלפה" או "swap", והוא מטרתה לשחרר מקום בזכרון. לשם כך, נוצרך להוסיף עוד מצבים ייחודיים לתהליכיים.

מימוש הפתרון:

ברגע שמערכת הפעלה יוצרת תמונה זיכרון לתהיליך, היא לא רק מעלה אותה לזכרון אלא גם מעתיקה אותה לדיסק. כך לכל תהיליך יש שתי תמונות - בזכרון הראשי ובדיסק. ברגע שיש צורך ב-swap, המערכת פשוט מעדכנת את החלקים שהשתנו בזכרון הראשי.

מה קורה בזמן השעה?

כאשר התהיליךמושחה וועובר לדיסק, הוא נכנס למצב "Blocked-Suspended". זהו מצב בו המערכת הפעלה צריכה להחזיר את התמונה לזכרון הראשי. ברגע שהטהיליך מוחזר לזכרון, הוא משנה את מצבו ל"-Ready". "Suspended

הערה:

מערכת הפעלה משמרת את ערך ה-PC של תהיליך המקורי במבנה הנתונים המייצג את התהיליך.

שאלה: האם פעולה השעה היא פעולה יקרה?

תשובה: לא בהכרח. בעצם היא יחסית זולה מבחינה משאבים.

שאלה: האם אפשרי ומומלץ לטעון את הזיכרון ללא תוכן?

תשובה: כן. מערכת הפעלה תטען את דפי התהיליך לפי הצורך (on demand). זה חסוך זיכרון וזמן גישה לדיסק. במרבית הזמן, ה-DRAM יהיה בדיסק ולא ב-RAM.

בשלב הבא, נדון בשאלת כיצד המעבד יודע שלא כל תמונה התהיליך נתענה לזכרון הראשי.

דוגמא מ-Unix

מתי מתבצעת פעולה ההחלפה?

- א. כאשר נאמר הזיכרון לקרנל.
- ב. כאשר אין מספיק מקום ליצירת תהיליך בן באמצעות `fork()`.
- ג. כאשר אין מספיק מקום להרחבת ה-heap באמצעות `brk()`.
- ד. כאשר אין מספיק מקום להרחבת ה-stack.
- ה. כאשר תהיליך במצב ready עם עדיפות גבוהה נמצא בדיסק.

אילו תהילכים בדרך כלל מוחלפים?

- א. תהילכים במצב READY שהוחלפו יש להם עדיפות גבוהה.
- ב. תהילכים גדולים בעדיפות נמוכה שהוחלפו.

הערות והסבירים נוספים:

- א. הפקודה `brk` משמשת להרחבת ה-heap. הפעולה של היצירה של התהיליך יכולה להיות יקרה, אך ה-`suspend` הופך לפחות יותר.
- ב. מערכת הפעלה יכולה להחליט להוריד לדיסק רק את החלק של התהיליך שבאמת השתנה.
- ג. **טיפ ליעול קוד:** אם אתם מעוניינים בתוכנה מהירה יותר - טנו לכתוב פקודות צמודות לתוך אותו ה-SECTION המתאים.

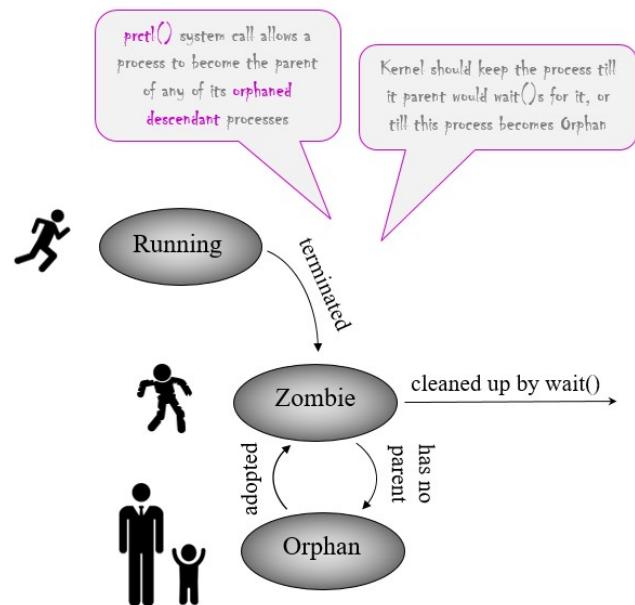
בالمושך, עלינו לש拷ול את השיטות בהן המערכת הופכת להחליף תהילכים, כולל תהילכים גדולים בעדיפות שונה. המטרה היא ליעול את הביצועים ולודוא שהמערכת מגיבה בצורה הטובה ביותר לדרישות המשתמש.

Terminated processes

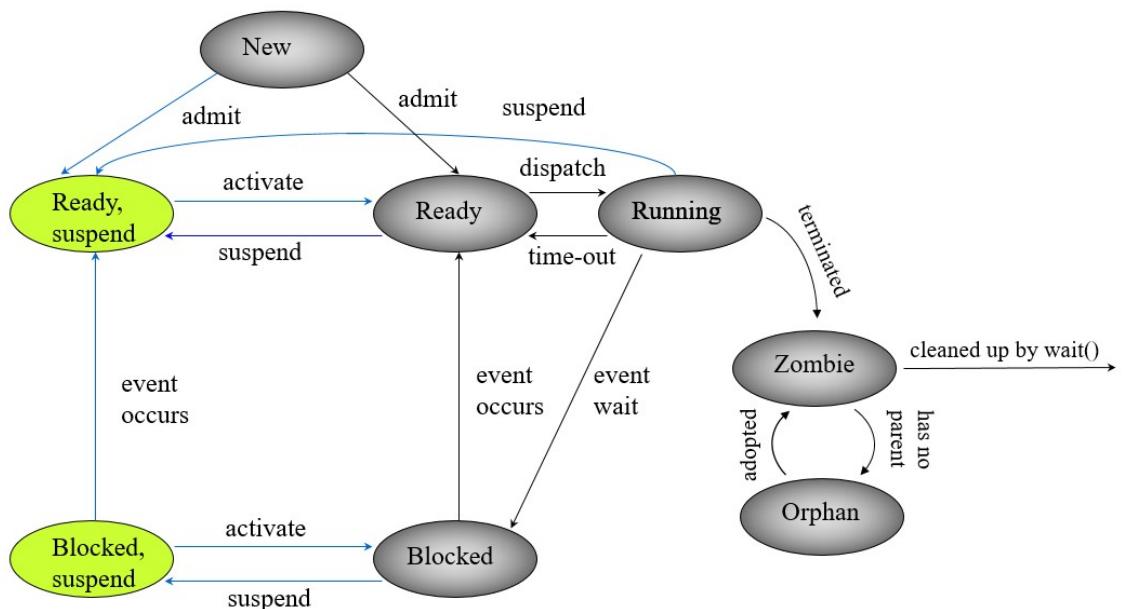
- ❑ **Zombie** – process which is terminated but still holds PCB
 - PCB is erased by kernel when process' parent executes `wait()`

- ❑ **Orphan** – process which parent is terminated

- ❑ **Init process adopts Orphan processes**
 - Init locates Orphan Zombies and `waitpid()`s for them, allowing OS then to deallocate their resources



Process State Diagram with Orphans



סיכום התהילה

סיבות לסיום התהילה

- . יציאה רגילה, לדוגמה באמצעות `exit` בוונדוס (יציאה יזומה של התהילה).
- . שגיאה, כשההתהילר מנסה להריץ קובץ שאינו קיים (יציאה יזומה של התהילר).
- . שגיאה קריטית בביצוע התהילר כמו הרצת הוראה לא חוקית (יציאה לא יזומה של התהילר).
- . הריגה באמצעות `kill` בוונדוס (לא יזומה של התהילר).

לשם תמייה במצב סיום התהילר יש לנקות בחשבון מצבים בהם התהילר בן של התהילר אחר או שהוא בעצם התהילר אב של תהליכי אחרים.

א. זומבי: תהליך שהסתיים אך עדין מחזיק ב-PCB. הליבה מסירה את ה-PCB כאשר האב של התהליך מבצע

. `() wait`

ב. יתום: תהליך שאבוי הסטים לפניו, במצב זה התהליך הראשון (`Init`) מאמץ את היתומים, מבצע `waitpid()` עבורם ומשחרר את משאבייהם.

שאלות ותשובות

שאלה: מה הפעולה שمبיאה את המידע על הבן?

תשובה: הפעולות הן `wait` ו- `waitpid`.

שאלה: למה יש מצב `exit`?

תשובה: כדי לנוקוט את התהליך מהזיכרון.

זהה התהליך

כל תהליך מזוהה באמצעות PID. האב שלו מזוהה באמצעות PPID. השימוש במספר זיהות מאפשר תקשורת בין תהליכים.

בעיה: אם תהליך בן שלוח לתהליך אב אותן בזמן שהאב כבר סיים, הוא כבר לא "האב האמיתי" ויתכן שהאות יגיע לתהליך אחר שכבר הוקצה באותו PID. למען הפתרון, ניתן להקטין את הסיכון באמצעות חילקה PID-ים בצורה משולבת ולא מידית.

עזרה: דיברתי על זה בחלוקת המתעסק בקריאות מערכות במסמכיו היסicos למעבדות..

(main) Fields of Process Control Block (PCB)

Process management	Memory management	File management
PID	Pointer to .text segment	Root directory
Parent process ID	Pointer to .data segment	Working directory
Process state	Pointer to Virtual Table	File descriptors table
Priority		User ID
PC (Program Counter)		Group ID
Stack Pointer		IO allocated devices
Registers		
Pending / blocked signals		
Creation timestamp		
CPU time used		

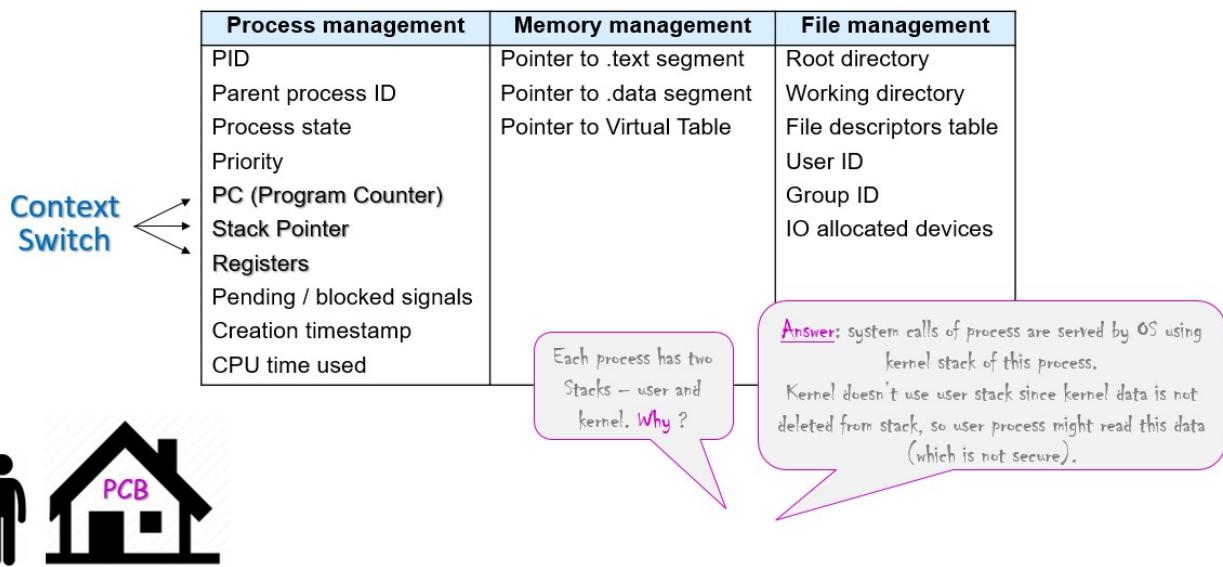


(main) Fields of Process Control Block (PCB)

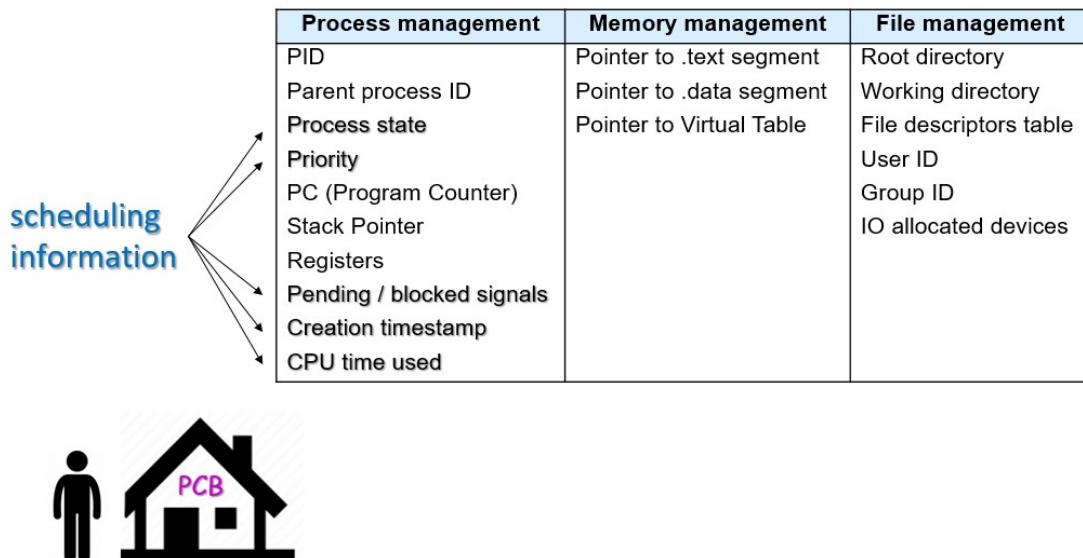
Process management	Memory management	File management
PID	Pointer to .text segment	Root directory ←
Parent process ID	Pointer to .data segment	Working directory ←
Process state	Pointer to Virtual Table	File descriptors table
Priority		User ID ←
PC (Program Counter)		Group ID ←
Stack Pointer		IO allocated devices
Registers		
Pending / blocked signals		
Creation timestamp		
CPU time used		



(main) Fields of Process Control Block (PCB)



(main) Fields of Process Control Block (PCB)



(main) Fields of Process Control Block (PCB)

Process management	Memory management	File management
PID	Pointer to .text segment	Root directory
Parent process ID	Pointer to .data segment	Working directory
Process state	Pointer to Virtual Table	File descriptors table
Priority		User ID
PC (Program Counter)		Group ID
Stack Pointer		IO allocated devices
Registers		
Pending / blocked signals		
Creation timestamp		
CPU time used		



process address
space information

(main) Fields of Process Control Block (PCB)

Process management	Memory management	File management
PID	Pointer to .text segment	Root directory
Parent process ID	Pointer to .data segment	Working directory
Process state	Pointer to Virtual Table	File descriptors table
Priority		User ID
PC (Program Counter)		Group ID
Stack Pointer		IO allocated devices
Registers		
Pending / blocked signals		
Creation timestamp		
CPU time used		



process
resources
information

מימוש תהליכיים

מודל התהליכיים מורכב ממספר רכיבים חיוניים כगון טבלת התהליכיים, מצבו הנוכחי של תהליכיים וניהול פסיקות. בפרק זה נדונו במימוש טבלת התהליכיים.

מערכת הפעלה מנהלת טבלת תהליכיים, שהיא מערך של מבנים שככל אחד מתוכם מייצג תהליך יחיד. המידע הזה מאפשר למערכת הפעלה לנוול את תהליכיים באופן יעיל.

בראש ובראשונה, כל רשומה בטבלה היא **Process Control Block (PCB)**, שמכילה מידע חיוני על התהליכיים:
א. **מצהירים:**

- מזהה התהיליך (PID).
- מזהה התהיליך האב (Parent process ID).

ב. מידע לגבי הוחלפה (Context Switch):

- PC (Program Counter)
- Stack pointer
- Registers

ג. מידע לגבי זמן:

זמן התהיליך (Creation time), עדיפות (Priority), מצב התהיליך (Process state), מותם או ממתינים (Deadline), הזמן בו התהיליך נוצר (Creation timestamp), הזמן המעבדה שהושתמש (CPU time used), זמן המעבדה שהושתמש (timestamp).

ד. מידע לגבי מרחב הכתובת של התהיליך:

- Pointer to .text segment
- Pointer to .data segment
- Pointer to Virtual Table

ה. מידע לגבי משאבי התהיליך:

טבלת הקבצים (File descriptor table), התקני ה-I/O שהוקצו.

במהלך הריצה, מערכת הפעלה יכולה לעדכן את ה-PCB לפי הצורך. לדוגמה, אם תהיליך צריך לבצע הדפסה, המערכת תעדכן את השדה המתאים.

בחינת הזיכרון, לכל תהיליך יש שתי מחסניות: אחת במרחב המשמש והשנייה במרחב הקרנל. הפרדה זו חיונית לשמירה על בטיחות המידע.

הערה: רעיון אחר חור על עצמו בקורס: אלגוריתם מסובך לא טוב, כי הוא לוקח יותר מדי זמן ויכול לדרש מבנים גדולים מדי. בהמשך הקורס נבחן איך הרעיון זה מושם באלגוריתמים שונים ואילו אתגרים הם מציגים.

Process Management : outline

❑ Process states and structures

❑ Process management

❑ Inter-process communication

❑ Signals

❑ Threads

❑ Specific implementations

When is a new process created?

1. OS Daemon processes ————— like page Daemon process
2. fork() system call ————— fork()
3. user request ————— run program on Shell

When does a process terminate?

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (**involuntary**)
4. Killed by another process (**involuntary**)

יצירת תהליכיים:

ישנם 3אירועים עיקריים בהם תהליכיים יכולים להיווצר: אתחול המערכת, הריצה של קריית מערכת ליצירת תהליכיים (fork), ובקשה מהמשתמש ליצירת תקין חדש.

במהלך אתחול המערכת, מספר תהליכיים נוצרים, כולל תהליכיים אשר מנהלים תקשורת עם המשתמש ותהליכי הרצים ברקע (daemon) אשר מספקים פונקציונליות ייחודית כמו אימייל או בקשות לדפי אינטרנט.

תהליך יכול להיווצר כאשר תקין אחר פונה למערכת הפעלה בבקשת תהליכיים על מנת לתמוך במשימות שלו. זה ייעיל כאשר העבודה יכולה להתפרק למספר תהליכיים **עצמאיים** אך בעלי קשר אחד לשני. במערכות אינטראקטיביות, משתמשים יכולים ליצור תקין על ידי הקלדת פקודה או לחיצה על לחצן.

ריצה של תהליך יכולה להסתיים כתוצאה ממספר תנאים, אשר ניתן לסייעו סיום ריצה מרצונו של התהליך או שאינו מרצונו של התהיליך.

סיכום ריצה מרצונו של התהיליך:

כאשר תהליך מסיים את המשימה ומסיים את ריצתו עצמאית. הדבר מתרחש בין אם הייתה ריצה תקינה (ואז הקומפイル הוסיף שורה קוד בבקשתו לסימן את ריצת התוכנית), ובין אם הייתה שגיאה בעקבותיה התהיליך יזם עצמאית בקשה לסימן את ריצת התהיליך (לדוגמה: תהליך בקש לפתוח קובץ, הקובץ לא קיים, ולכן יש תנאי בקוד של התוכנית לסייעו סימן את הריצה).

סיכום ריצה שאינו מרצונו של התהיליך:

יכול להתרחש עקב שגיאה שנוצרה על ידי התהיליך עצמו, לרבות בגלל באג. זה כולל הוראות לא חוקיות, התייחסות לזמן שלא קיים או חילוק באפס. במערכות מסוימות, התהיליך יכול לשלוות אותן (סיגナル) למערכת הפעלה כדי לנוהל באופן עצמאי את השגיאות הללו, כתוצאה מכך התהיליך מקבל פסקה במקום לסימן את ריצתו.

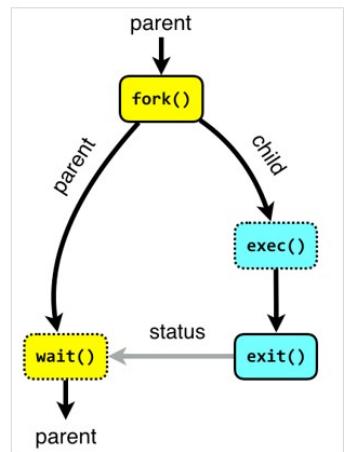
Process Creation (Unix)

- ❑ **PID = `fork()`** - create a child process
- ❑ **`execvp(name, args)`** – replace image by name, with arguments args
- ❑ **`exit(status)`**
- ❑ **`wait(status)` / `waitpid(pid, status)`** - wait for termination of a (specific) child process; status argument is a pointer to a variable that would hold child process termination status

Process may use `getpid()` system call to get its PID.

In Windows OS, `fork() + exec()` is a single system call. What are the advantages and disadvantages of this?

Answer: between `fork()` and `exec()` we can configure child process. For example, we can redirect its `stdin/stdout`. But we pay for two system calls, which means we pay double context switch.



מנגנון יצירת תהליך

בכל המקרים שצוינו לעיל, תהליך חדש נוצר בעזרת קריאת מערכת. קריאה זו מנהה את מערכת הפעלה ליצור תהליך חדש ומציין איזו תוכנית התהיליך החדש צריך להריץ.

בUNIX, קריאת המערכת `fork` מייצרת עותק זהה של תהליך אחר, כולל את אותה תמונה הזיכרון וקובציים פתוחים. בדרך כלל תהליך הבן יריץ הוראות כגן `execve` כדי לשנות את תמונה הזיכרון ולהריץ תוכנית חדשה. בכך תהליך יכול לבצע מניפולציות ב-`fd` כמו `redirect.io`.

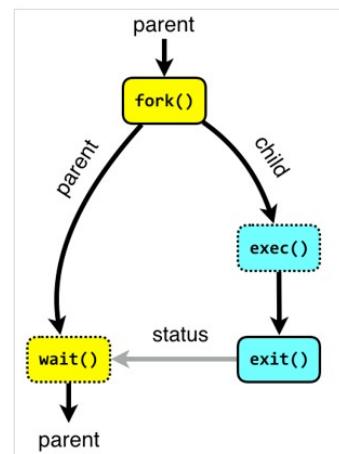
אם צריך לחתה העדפה בזמןון לבן? כן. (תשובה במצגת)
הערה: הסברתי על תהליך ה-`fork` בצורה מוגברת בסיכון שלי במעבדות. במידה יהיה צורך בהרחבות נוספות,

Process Creation (Unix)

- ❑ PID = `fork()` - create a child process
- ❑ `execvp(name, args)` – replace image by name, with arguments args
- ❑ `exit(status)`
- ❑ `wait(status)` / `waitpid(pid, status)` - wait for termination of a (specific) child process; status argument is a pointer to a variable that would hold child process termination status

After fork, should OS Scheduler prefer to run child process? Why?

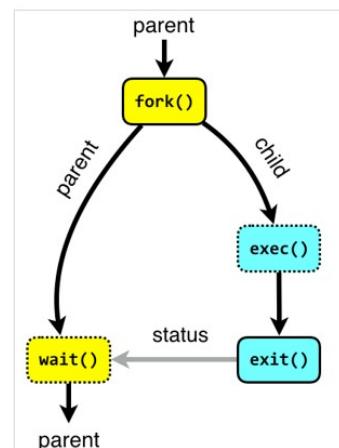
Answer: parent address space is marked as Copy-On-Write (COW). If we run parent process first, all the memory changed by parent would be copied to child address space. This might be **wasteful**, since child process probably `exec()`s and thus would not use this copied data.



Process Creation (Unix)

- ❑ PID = `fork()` - create a child process
- ❑ `execvp(name, args)` – replace image by name, with arguments args
- ❑ `exit(status)`
- ❑ `wait(status)` / `waitpid(pid, status)` - wait for termination of a (specific) child process; status argument is a pointer to a variable that would hold child process termination status

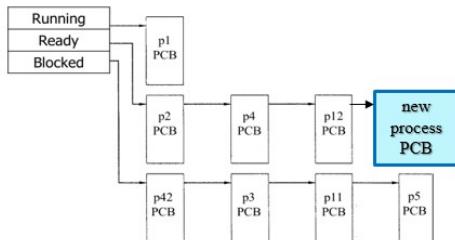
Note: file descriptors table is copied from parent process to child process. This means that all opened files of parent process are also opened in child process. If we want to close them before child process `exec()`s, we can do this by system call `fcntl()` with `FD_CLOEXEC` flag on.



Process Creation

- ❑ Find free Process Table entry (PCB) and copy parent's PCB info into it
 - PCB entry number is indeed process identifier (PID)
- ❑ Allocate initial memory for process Stacks (user and kernel)
- ❑ **Mark all parent's memory as "copy on write" (COW)**
 - possibly suspend some process(es) for this
- ❑ Put the process to (appropriate) *scheduling queue*

If Process Table is full, OS returns error.



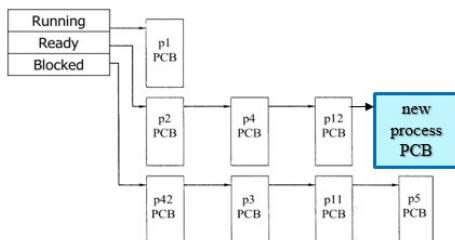
Seems like fork() should be expensive, but indeed it works very fast. **Why?**

Answer: address space is Copied-On-Write (COW), means that we needed do not copy parent address space, until one of parent or child tries to modify shared process image.

Process Creation

- ❑ Find free Process Table entry (PCB) and copy parent's PCB info into it
 - PCB entry number is indeed process identifier (PID)
- ❑ Allocate initial memory for process Stacks (user and kernel)
- ❑ **Mark all parent's memory as "copy on write" (COW)**
 - possibly suspend some process(es) for this
- ❑ Put the process to (appropriate) *scheduling queue*

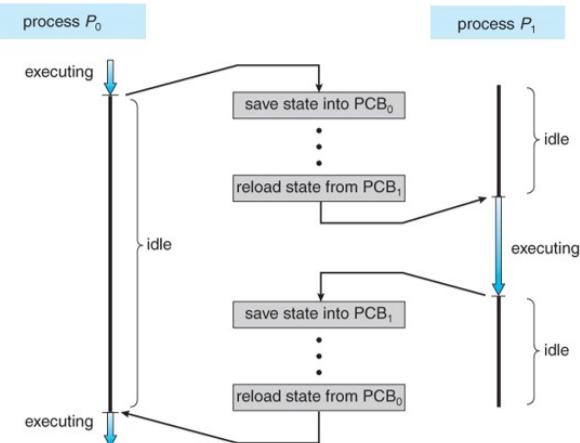
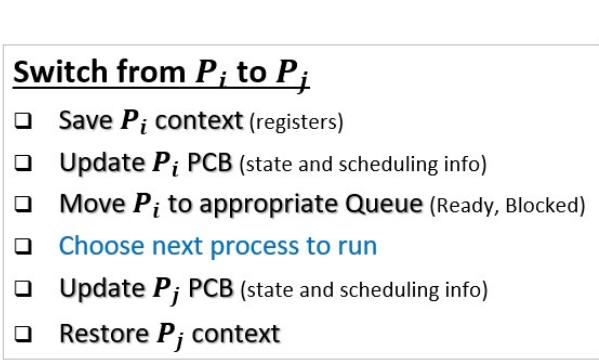
If Process Table is full, OS returns error.



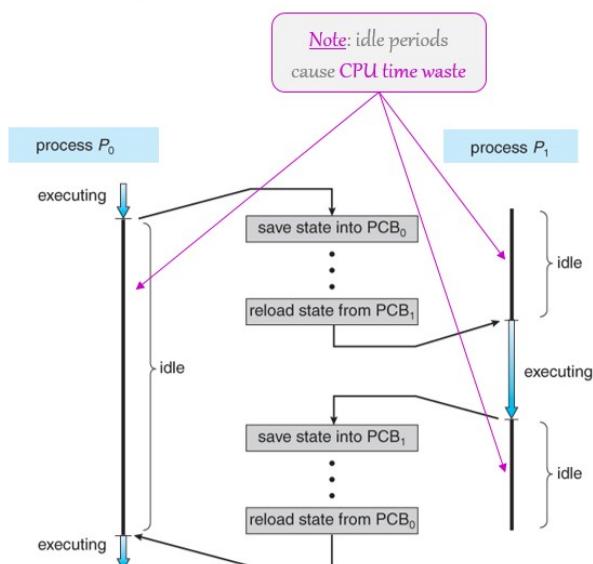
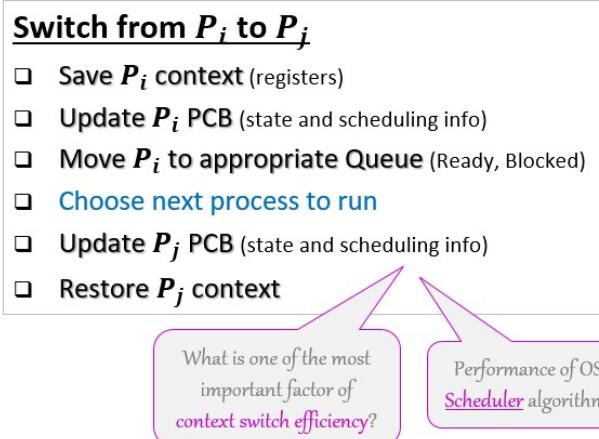
NO. Maybe some other process P2 still keeps this PID, and might use it, for example, for communication with P1. Thus, communication would be with the new process, which is wrong behaviour.

Assume that some process P1 is just terminated. Should OS **immediately** give its PID to some new process?

Process Context Switch



Process Context Switch



העלות של context switch היא גבוהה.

שאלה: מתי מתבצע החלפת הקשרים?

תשובה: זה קורה כאשר CPU מרים את ה-scheduler, או כאשר הוא מתפנה או כאשר הקוונטים זמן מסתיים. ככל שמערכת הפעלה מחליטה לעיתים קרובות יותר לבצע החלפת הקשרים אנחנו רק נפסיד, כי זה לוקח זמן ריצה לטובות חישובים של מערכת הפעלה.

כדי להסביר את זאת נשתמש בדוגמה מהח'רים האמתיים: אנחנו לחנות לינות מוצרים ובמקום שהמוכרת תמכור לנו מוצרים היא מסדרת את החנות כדי שהיא נוח יותר. זה נחמד אבל זה על חשבון הזמן שלנו.

אם נסכם, הסוגיה שמערכת הפעלה צריכה לפתור היא האם לעשות לעיתים רחוקות או לעיתים קרובות?

מבחן על .context switch
לעתים רחוקות: המשתמש ירגע את זה.
מיشهו פה לא יהיה מרצו.
בשביל זה נרצה את האלגוריתם הכי קצר

Process Management : outline

- ❑ Process states and structures
- ❑ Process management
 - ❑ Inter-process communication
 - ❑ Signals
 - ❑ Threads
 - ❑ Specific implementations

תקשורת בין תהליכיים

Inter-Process Communication

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void* create_shared_memory(size_t size) {
    // readable and writable:
    int protection = PROT_READ | PROT_WRITE;
    int visibility = MAP_SHARED | MAP_ANONYMOUS;
    return mmap(NULL, size, protection, visibility, -1, 0);
}
int main() {
    char parent_message[] = "hi";
    char child_message[] = "bye";
    void* shmem = create_shared_memory(128);
    memcpy(shmem, parent_message, sizeof(parent_message)); // parent writes message
    int pid = fork();
    if (pid == 0) {
        printf("child read: %s\n", (char*)shmem);
        memcpy(shmem, child_message, sizeof(child_message)); // child writes message
    } else {
        wait(NULL);
        printf("parent read: %s\n", (char*)shmem);
    }
    return 0;
}
```

Shared memory is the **fastest** way of inter-process communication. Synchronization tool is needed to avoid race conditions.

shared buffer would be readable and writable

anonymous means that no one except the process and its children can use the buffer

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in **addr**, the size specifies the size of the mapping. If **addr** is **NULL**, then the kernel chooses the (page-aligned) address at which to create the mapping;

Other communication tools:

- Pipes
- Signals
- Files
- Sockets

```
marina@vm:~/OS $ gcc -m32 main.c && ./a.out
child read: hi
parent read: bye
marina@vm:~/OS $
```

הכל האולטימטיבי לתקשורת בין תהליכיים: זיכרון משותף.
יש זיכרון ש-2 תהליכיים יכולים לגשת אליו. צריך בשבייל זה סינכרוניזציה. אין סייגלים או משהו אחר בכלל. יש רק אישור ראשוני וזהו. מכאן והלאה מערכת הפעלה לא מתערבת חוץ מקרה אחד:

סנכרון. כלומר נשתמש למערכת הפעלה על מנת להשתמש בכל הsnsכרון שלו. משתמשים רבים בתעשייה, במיוחד ב-back-end. כל פעם שמקשים שירות ממערכת הפעלה יש חלפת הקשרים.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void* create_shared_memory(size_t size){
    //readable and writeable:
    int protection = PROT_READ | PROT_WRITE;
    //the shared memory will be both readable and writeable.
    int visibility = MAP_SHARED | MAP_ANONYMOUS;
    //ANONYMOUS - only the parent process and it's children can use the
buffer.
    return mmap(NULL, size, protection, visibility, -1, 0);
    //create a new mapping in virtual address space in the address which is
the first argument. since it is NULL, the kernel will choose the address.
}
```

בשלב זהה האדרנו מעין בנאי ליצירת מיפוי הזיכרון המשותף ל-2 התהליכים. כתע נראה כיצד השתמש בו:

```
int main(){
    char parent_message[] = "hi";
    char child_message[] = "bye";
    void* shmem = create_shared_memory(128);
    memcpy(shmem, parent_message, sizeof(parent_message));
    //parent writes the message.
    if(fork() == 0){
        //the code that the children will run:
        printf("child read: %s\n", (char*)shmem);
        //since the string is null-terminated, and since memcpy wrote the
message from the start of the mapping, it will print "hi".
        memcpy(shmem, child_message, sizeof(child_message));
        //the child writes a response to it's parent.
    }else{
        //parent
        wait(NULL); //wait for it's single child to terminate.
        printf("parent read: %s\n", (char*)shmem);
        //the parent will print the response message.
    }
}
```

```
    return 0;  
}
```

לא חובה לדעת את זה בקורס, אנחנו לא מחייבים כאן את ה-SPL כמו ב-LT.

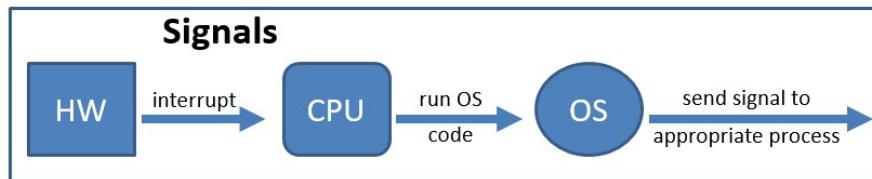
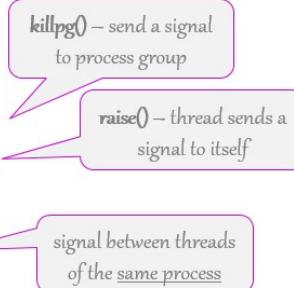
Process Management : outline

- ❑ Process states and structures
- ❑ Process management
- ❑ Inter-process communication
 - ❑ Signals
 - ❑ Threads
 - ❑ Specific implementations

Unix signals

- ❑ signal is a **software interrupt**

- ❑ signals are generated:
 - from keyboard: **Ctrl-C, Ctrl-Z, ...**
 - from command line: **kill -<sig> <PID>**
 - from code: **kill(PID, sig)**
 - from code: **pthread_kill(TID, sig)**



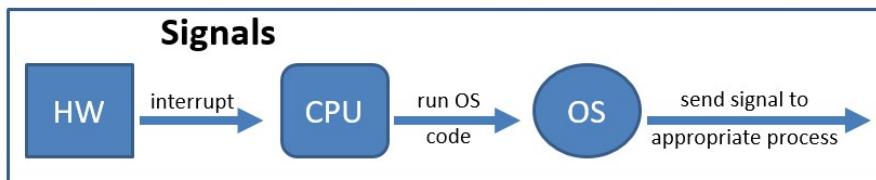
Unix signals

- ❑ signal is a **software interrupt**

- ❑ signals are generated:

- from keyboard: **Ctrl-C, Ctrl-Z, ...**
- from command line: **kill -<sig> <PID>**
- from code: **kill(PID, sig)**
- from code: **pthread_kill(TID, sig)**

A thread can obtain the set of signals that it currently has pending using **sigpending()**. This set will consist of the union of the set of pending process-directed signals and the set of signals pending for the calling thread.



Unix signals

```
marina@vm:~/SPLab $ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT
4) SIGILL      5) SIGTRAP     6) SIGABRT
7) SIGBUS      8) SIGFPE     9) SIGKILL
10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGNALRM   15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT
19) SIGSTOP    20) SIGTSTP    21) SIGTTIN
22) SIGTTOU   23) SIGURG     24) SIGXCPU
25) SIGXFSZ   26) SIGVTALRM  27) SIGPROF
28) SIGWINCH  29) SIGIO      30) SIGPWR
31) SIGSYS
```

Each signal has a **disposition**, which determines how the process behaves when it is delivered the signal.

Term – terminate the process
Ign – ignore the signal
Core – terminate the process and dump core
Stop – stop the process
Cont – continue the process if it is currently stopped

Most of signals can be ignored, blocked or handled.

- **SIGKILL** – kill process (can't be blocked, ignored, or caught by handler)
- **SIGSTOP** – stop process (CPU stops executing the current instruction of the process), but does not kill the process (can't be blocked, ignored, or caught by handler)

סיג널ים במערכת UNIX

סיגナル הוא פסיקה ברמת התוכנה, נקרא גם פסיקה תוכניתית. הוא יכול להיווצר ממספר סוגים של גורמים: פסיקה חומרתית או קריית המערכת **KILL**. ניתן ליצור סיגナル גם באמצעות המקלדת, דרך הפקודה במסוף, או דרך הקוד עצמו.

דרכים ליצירת סיגナル

- ממקלדת, לדוגמה: Ctrl-C, Ctrl-Z משורת הפקודה, כמו:
- דרך קוד בשפת C:

- `kill(PID, sig)`

- `pthread_kill(TID, sig)`
- `killpg()` שולח סיגナル לקבוצת התהליכים:
- `raise()` ת'רד שולח לעצמו סיגナル:

ג. סיגנלים בין ת'רדים של אותו התהליך

אשר מרים קוד של מערכת הפעלה, בעורתה, בזיהוי interrupt. דרך החומרה: החומרה פונה לمعالג, באמצעות שולחת סיגナル לתהליכי המתאים.

מאפייני הסיגナル

כל ת'רד יכול לקבל את הסיגנלים המממשנים לו באמצעות הפקודה `sigpending()` לכל סיגナル יש התנהלות, או כינויים אחרים: `disposition` ההתנהלות הזו קובעת איך התהליך מגיב כאשר הסיגナル מתתקבל:

- סיום התהליכי: Term
- התעלמות מהסיגナル: Ign
- Core dump סיום התהליכי ויצירת core
- עצירת התהליכי: Stop
- המשך התהליכי אם הוא עצר: Cont

חשוב לדעת שלרוב הסיגנלים אפשר להתעלם, לחסום או לטפל בהם אך ישנים סיגנלים כמו `SIGKILL` ו- `SIGSTOP` שלא ניתן לחסום או להתעלם מהם קריאת המערכת `signal` מאפשרת לנו להגדיר ההתנהלות מותאמת אישית לסיגנלים.

Handling signals

- `signal(signum, [handler] | SIG_IGN | SIG_DFL);`

signal number

signal handler

ignore signal

take OS default action

signal() defines an action to be taken to treat a signal

signal() may **reset** the signal action back to SIG_DFL. Thus we should reinstall handler. This opens up a window of vulnerability between the time when the signal is detected and the handler is reinstalled during which if a second instance of the signal arrives, the default behavior (usually terminate) occurs.

The signal disposition is a **per-process attribute**: in a multithreaded application, the disposition of a particular signal is the **same for all threads**.

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

void handler(int signum) {
    signal(SIGINT, handler);
    printf("caught Ctrl+C signal\n");
}

int main() {
    signal(SIGINT, handler);
    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
    return 0;
}
```

```
0
1
2
(Ctrl+C)
caught Ctrl+C signal
3
4
(Ctrl+C)
caught Ctrl+C signal
5
6
7
8
(Ctrl+Shift+)
Quit (core dumped)
```

Handling signals

- `signal(signum, [handler] | SIG_IGN | SIG_DFL);`

signal() returns the old action for treating the signal

we can recognize that we try to change a disposition of ignored signal, and continue ignoring it

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

void handler(int signum) {
    signal(SIGINT, handler);
    printf("caught Ctrl+C signal\n");
}

int main() {
    if (signal(SIGINT, handler) == SIG_IGN)
        signal(SIGINT, SIG_IGN);
    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
    return 0;
}
```

```
0
1
2
(Ctrl+C)
caught Ctrl+C signal
3
4
(Ctrl+C)
caught Ctrl+C signal
5
6
7
8
(Ctrl+Shift+)
Quit (core dumped)
```

Handling signals

- `signal(signum, [handler] | SIG_IGN | SIG_DFL);`

```
A child created via  
fork() inherits a copy of  
its parent's signal  
dispositions.
```

```
During an execve(),  
the dispositions of handled signals  
are reset to the default; the  
dispositions of ignored signals  
are left unchanged.
```

```
0  
1  
2  
(Ctrl+C)  
caught Ctrl+C signal  
3  
4  
(Ctrl+C)  
caught Ctrl+C signal  
5  
6  
7  
8  
(Ctrl+Shift+)\nQuit (core dumped)
```

Handling signals

- `signal(signum, [handler] | SIG_IGN | SIG_DFL);`

```
A child created via fork()  
initially has an empty  
pending signal set.
```

```
The pending signal set  
is preserved across an  
execv().
```

```
0  
1  
2  
(Ctrl+C)  
caught Ctrl+C signal  
3  
4  
(Ctrl+C)  
caught Ctrl+C signal  
5  
6  
7  
8  
(Ctrl+Shift+)\nQuit (core dumped)
```

טיפול בסיגנלים

במערכת הפעלה, סיגנל הוא אמצעי לספק התראה לתחילה על אירוע מסוים. הפונקציה `signal()` מאפשרת להגדיר אילו פעולות יתבצעו בתחילה כאשר הסיגנל מתתקבל.

הגדירה:

```
signal(signum, [handler | SIG_IGN | SIG_DFL]);
```

הפרמטרים:

- `signum`: מספר הסיגナル.
- `handler`: פונקציה לטיפול בסיגナル.
- `SIG_IGN`: הפעולות מהסיגナル.
- `SIG_DFL`: הפעולות בירית המחדל של המערכת הפעלה.

דוגמה לקוד המטפל בסיגナル C`Ctrl+C`:

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

void handler(int signum) {
    signal(SIGINT, handler);
    printf("caught Ctrl+C signal\n");
}

int main() {
    signal(SIGINT, handler);
    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
    return 0;
}
```

במהלך ריצת הקוד, כאשר המשתמש לוחץ על `Ctrl+C`, הודעה "caught Ctrl+C signal" מודפסת.

הערות:

- א. הפונקציה `signal()` יכולה לאפס את הפעולה של הסיגナル אל `SIG_DFL`. כדי לטפל זהה, אנו מתקנים מחדש את הטיפול בסיגナル בכל פעם שהטיפול בסיגナル מתבצע.
- ב. כאשר קורה `fork()`, הילד מקבל עותק של הטיפול בסיגנלים של ההורה.

מטפלי הסיגנלים:

פונקציות אלה הן פונקציות המוגדרות על ידי המתכנת, המבצעות כאשר סיגナル מגיע לתהילר.

התנהגות של הקרנל במעבר למרחב המשתמש:

במעבר מרחב הקרנל למרחב המשתמש, הקרנלבודק האם קיימים סיגנלים (שאינם `unblocked`) הממתינים לטיפול. במידה וכן, הקרנל מתכוון להריץ את הפונקציה, בונה לה `activation frame` בסSTACK, ו מעביר את השילטה חוזרת למרחב המשתמש.

טיפול בקריאה מערכת:

אם הפונקציה הופעלה במהלך קריאת מערכת שהייתה נחסמה, הקריאה תוחזר לאחר שהפונקציה תסתיים, או תכשל.

התמודדות עם קישור הסיגנלים:

קיימת בעיה בנווגע להתקנות הקריאה – אם הקישור הוא חד פעמי או לא. מכיוון שההתקנות היא תליה במוגבלות רבות, **הופסק השימוש בקריאה זו**.

במוקם, מומלץ להשתמש בקריאה המערכת `sigaction` – הקישור ישאר עד שיתקבל הוראה מפורשת לבטולו. השימוש ב`signum` מתבצע כאשר ה-`handler` משמש לטיפול במספר סיגנלים.

דוגמאות:

א. התקבל סיגナル והתהיליך הפעיל את הפונקציה, لكن הциום בוטל. אם התקבל סיגナル אחר לפני החידוש של הциום, המערכת תפעל לפי ברירת המחדל.

ב. לפונקציה יש ערך המוחזר. הערך זהה מצין איזו פונקציה טיפולה בסיגナル לפני השינוי.

שיקולים בעקבות `execvp` ו-`fork`:

א. לאחר ביצוע `fork`, כל הסיגנלים שהתקבלו עד אז והагדרות הקודמות נשארות ללא שינוי. אבל, ברגע שמתבצעת `execvp`, ההגדרות מוחזרות להתקנות ברירת המחדל. זאת מושם שלאחר ביצוע `execvp` ה-

process image נמחק וכך גם ה-`handlers` מתמחקים.

ב. קבצים אינם חלק מה-`process image`.

ג. מערכת הפעלה אינה חייבת להגיב מיד לסיגナル שהתקבל. הסיגנלים שנשלחו אך לא התקבלו יישלחו ל-PCB של התהיליך.

Handling signals

❑ `sigaction(signum, new_action, &old_action);`

`sigaction()` is Linux system call that defines an action to be taken to treat a signal

the previous action is saved in `old_action`

necessary provided (but in our case, unused) arguments

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

void handler(int signum, siginfo_t* x, void* y) {
    printf("caught Ctrl+C signal\n");
}

int main() {
    struct sigaction new_action, old_action;
    memset(&new_action, 0, sizeof new_action);
    memset(&old_action, 0, sizeof new_action);
    new_action.sa_sigaction = handler;
    sigaction(SIGINT, &new_action, &old_action);
    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
    return 0;
}
```

```
0
1
2
(Ctrl+C)
caught Ctrl+C signal
3
4
(Ctrl+C)
caught Ctrl+C signal
5
6
7
8
(Ctrl+Shift+)
Quit (core dumped)
```

`sigaction()` can block other signals until handler returns, `signal()` not always can do this

`sigaction()` does not reset the signal action

קריאה מערכת להגדרת התקנות לסיגנלים – `sigaction`

הקריאה `sigaction` מאפשרת לתהיליך לשנות את הפעולה אשר יש לבצע כאשר מקבל סיגナル.

ההגדרה של `sigaction`

```
int sigaction(int signum, const struct sigaction *act, struct sigaction  
*oldact);
```

כארה:

- **signum**: מספר הסיגナル אשר תגיב לו הfonקציה.
- **act**: הפעולה החדשה לסיגナル שמספרו **signum**.
- **oldact**: מצביע שמשמש לשמירה של הפעולה הקודמת לסיגナル.

מבנה ה-**sigaction**

```
struct sigaction {  
    void (*sa_handler)(int); // הפעולה המקורית לסיגנל  
    void (*sa_sigaction)(int, siginfo_t *, void *); // פונקציה שמקבלת  
    /* שלושה ארגומנטים */  
    sigset_t sa_mask; // סט של סיגנלים לחסימה  
    int sa_flags; // סט של דגלים להפעולה  
    void (*sa_restorer)(void); // פונקציה לשחזר המצב הקודם  
};
```

אם SIGINFO Dolק אז ה-**sa_sigaction** הוא פונקציה.
ההchnerה מהקריאה היא 0 בהצלחה, ו-1 בכישלון.

יתרונות **signal** על פני **sigaction**

- הקריאה **sigaction()** היא קריאת מערכת בLINOKS להגדרת פעולה לסיגナル.
- הקריאה **(signal)** היא מוגדרת בתקן C אך לא בתקן POSIX - מכאן חוסר הנוימות שלה.
- באמצעות **sigaction()**, ניתן לחסום סיגנלים אחרים עד שה-handler מוחזיר שליטה. ב-**signal()** זה לא תמיד אפשרי.
- ב-**signal()** אינה מאפשרת את הפעולה לסיגナル.

דוגמת קוד

```
#include <stdio.h>  
#include <signal.h>  
#include <unistd.h>  
#include <string.h>  
  
void handler(int signum, siginfo_t* x, void* y) {  
    printf("caught Ctrl+C signal\n");  
}  
  
int main() {  
    struct sigaction new_action, old_action;  
    memset(&new_action, 0, sizeof(new_action));  
    memset(&old_action, 0, sizeof(old_action));
```

```

new_action.sa_sigaction = handler;
sigaction(SIGINT, &new_action, &old_action);

int i = 0;
for (;;) {
    printf("%d\n", i++);
    sleep(1);
}
return 0;
}

```

הפרמטרים `siginfo_t* x, void* y` הם ארגומנטים הדרושים אך אין בהם שימוש במקרה זה.

User-defined Signals

This implementation of `sig_usr` is **unsafe**. Why?

`printf()` is **not reentrant**, means, we cannot call `printf()` during execution of `printf()`. It uses global variables, which would be overwritten by the second call, which leads to wrong execution of the first call when go back to it.

```

void sig_usr(int SIG) {
    switch (SIG) {
    case SIGUSR1:
        printf("received SIGUSR1");
        break;
    case SIGUSR2:
        printf("received SIGUSR2");
        break;
    }
}

```

`SIGUSR1` and `SIGUSR2` are signals for **inter-process communication**. The default action is to terminate the process.

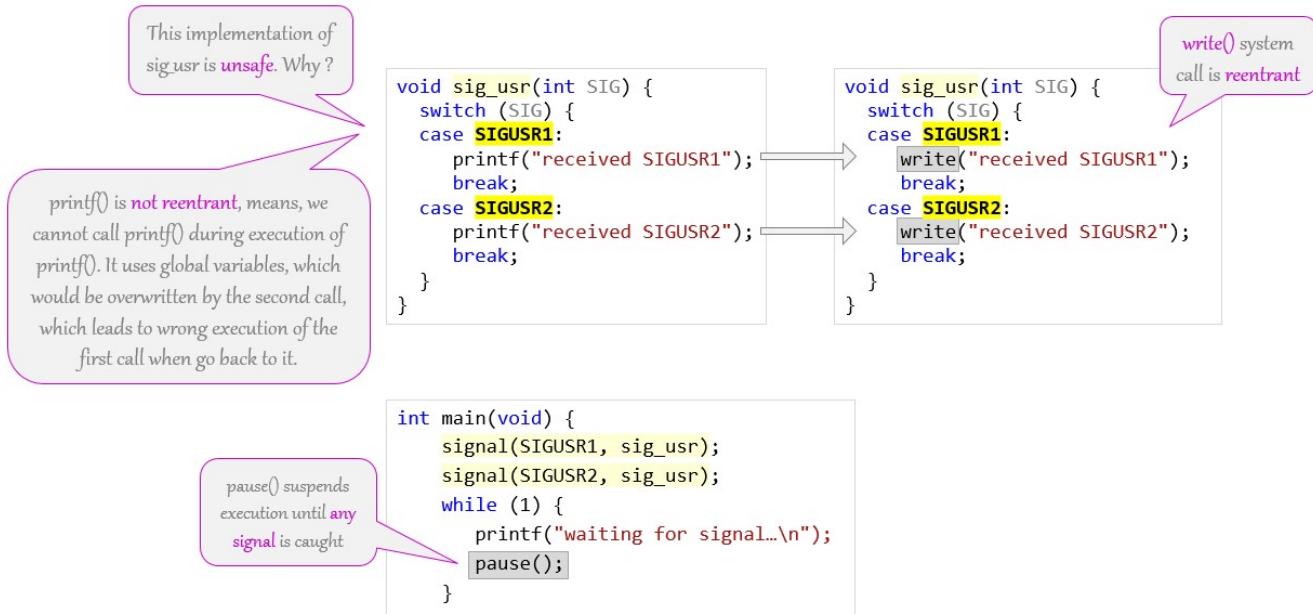
`pause()` suspends execution until **any signal** is caught

```

int main(void) {
    signal(SIGUSR1, sig_usr);
    signal(SIGUSR2, sig_usr);
    while (1) {
        printf("waiting for signal...\n");
        pause();
    }
}

```

User-defined Signals



סיגנלים מוגדרים על ידי המשתמש

בלינוקס ישנו שני סיגנלים מיוחדים שמתוכננים לתקשרות בין תהליכי, כך שהמשתמש יוכל לעשות בהם שימוש מבלי לדרכם הtentativeיות של טיפול בסיגנלים אחרים: **SIGUSR2** ו- **SIGUSR1**. בירית המחדל לשני הסיגנלים היא להרוג את התהליך.

אפשר להגדיר את הtentativeות התהילה' לאותם הסיגנלים באמצעות הפונקציה **:sig_usr**:

```
void sig_usr(int SIG) {
    switch (SIG) {
        case SIGUSR1:
            printf("received SIGUSR1");
            break;
        case SIGUSR2:
            printf("received SIGUSR2");
            break;
    }
}
```

בקוד הבא, התהילה' מחייבת לקבלת סיגナル:

```
int main(void) {
    signal(SIGUSR1, sig_usr);
    signal(SIGUSR2, sig_usr);
    while (1) {
        printf("waiting for signal...");
        pause();
    }
}
```

```

    }
}

```

הfonkziah pause () משעה את ביצוע התהיליך עד שייתקבל סיגナル.

הבעיות בקוד והצעת פתרון

הקוד הנ"ל הוא לא בטוח. הסיבה היא שהfonkziah printf () איננה "reentrant". ככלומר, אי אפשר לקרוא לה באופן שמשלב מספר תהליכי מבלי להכנס לערך הפעולה הראשונה לכאוס. הסיבה לכך היא שהוא משתמש במסתנים גלובליים, שבמיהה ויתבצעו שינויים לאותם המסתנים במהלך הקריאה השנייה, יכולים להוביל לתוצאה לא צפואה בהקריאה הראשונה.

פתרון לבעה זו הוא להשתמש בfonkziah write () שהיא ":"reentrant

```

void sig_usr(int SIG) {
    switch (SIG) {
        case SIGUSR1:
            write(STDOUT_FILENO, "received SIGUSR1", 17);
            break;
        case SIGUSR2:
            write(STDOUT_FILENO, "received SIGUSR2", 17);
            break;
    }
}

```

השימוש ב-write () מבטיח שהמסתנים הגלובליים לא ישתנו והפעולה היא בטוחה.

Unix Signals Data Structure

- Upon generating a signal, OS sets appropriate signal bit in PCB

On return from a system call or scheduling of a process onto CPU, OS checks whether there is a pending unblocked signal, and treats them.

A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked.

Signum	BLOCKED	handler	pending
1	false	SIG_DFL	0
2	false	SIG_IGN	0
3	false	func1	1
4	true	func2	0
5	true	SIG_DFL	1
...			

ignores signals would be dropped by OS

Signals may not be treated immediately. Yet not treated signal is called pending.

If a signal is generated multiple times till process gets it, this signal is delivered only once.

במערכת הפעלה אוניברסיטאית, המרכיבת הפעלה מגדירה את הסיגナル הרלוונטי ב-PCB. כאשר ישנה חרזה מפונקציית מערכת או בתהילר התזמון של התהילר לمعالג, המערכת הפעלה בודקת האם ישנים סיגנלים שלא נחסמו אך לא טופלו, ומטפלת בהם.

כאשר סיגナル נשלח לתהילר, הוא יכול להישלח לכל אחד מהת'רידים בתהילר שלא חסם את הסיגナル.

במבנה הנתונים ישנה טבלה המכילה את הערכים הבאים:

- signum
- BLOCKED
- handler
- pending

סיגנלים אשר מתעלמים מהם המערכת הפעלה מתעלמת ומשליכה אותם. ישנים סיגנלים שלא מטופלים מיד, וכאליה הנקראים "pending". אם סיגナル מסוים נוצר מספר פעמים עד שהטהילר מטופל בו, הסיגナル יועבר רק פעם אחת.

בכל ת'ריד בתהילר ישנה טבלה זו. ה-HANDLER הוא אחיד לכל הת'רידים, אך כל ת'ריד יכול להחליט כיצד להתייחס לסיגナル. הת'רידים הם חלק מהטהילר, וב-PCB ישנים מבנים השיכים לת'רידים. גם המערכת הפעלה עשויה לראות את הת'רידים והטהיליכים כמו שהוא או אפילו זהה, תלוי במשמעות הפרט של מערכת הפעלה.

בקצרה, כאשר סיגナル מגיע לתהילר, הוא מטופל על ידי ת'ריד אחד בתהילר.

Process Management : outline

- ❑ Process states and structures
- ❑ Process management
- ❑ Inter-process communication
- ❑ Signals
- ❑ Threads
- ❑ Specific implementations

ת'רידים (חותמים)

Processes \leftrightarrow Threads

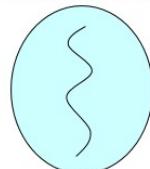
❑ thread is a basic unit of CPU scheduling

Why multi-threads and not multi-processes?

- faster creation
- faster context switch
- no flush of CPU caches
- faster communication
- shared environment and resources

process - common execution environment container

single-thread process



thread - code executor, local execution environment container

multi-thread process



Creation time: process \leftrightarrow thread

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

creation of process is **much costly** than creation of thread

ת'רדים (Processes) ותהליכיים (Threads)

ת'רד הוא יחידת בסיס לתזמון במעבד. ברוב המקרים, במקום ליצור מספר תהליכיים, כדאי ליצור ת'רדים רבים תחת תהליך אחד.

למה להשתמש בת'רדים מרוביים במקום בתהליכיים מרוביים?

- הם קלים יותר ליצירה והריסה.
- החלפת הקשרים מהירה יותר
- אין השפעה על מטמוני המעבד
- תקשות מהירה יותר, כולל שיתוף הפעולה בין מהיר ויעיל.
- קיים סביבה ומשאבים משותפים

הבדל בין ת'רדים לתהליכיים

תהליך הוא דומם - משהו שיש לו PCB, תמנונת זיכרון וכו'. לעומת זאת, הת'רד הוא החלק הביצועי של התהליך. כאשר אומרים שתהליך "רץ", הכוונה היא שהת'רד הראשי בתוך התהליך רץ.

דוגמאות לשימוש בת'רדים

א. **מעבד תמלילים:** מעבד התמלילים יכול להציג מסמר בזמן אמיתי. אם משתמש מוחק שורה מתוך ספר, ת'רד אחד יכול להתמקד במשתמש בזמן שני מבצע שינויים בקובץ.

ב. **עיבוד מידע:** באפליקציות שצרכו לערוך המון מידע, ת'רד אחד יכול להתמקד בקלט, השני בעיבוד הקלט והשלישי בפלט.

שאלה: אם יש לי COW (Copy On Write) - שהשכפול לא עמוק - למה זה עדין עולה יקר?

תשובה: כי ישנו overhead חדש. ב-COW, המידע מוגדר כ-**ONLY READ**, והסימונים האלה עלולים יקר. המשאב העיקרי הולך לכך, וזה גם קשור לזיכרון הוויירטואלי.

Threads in POSIX

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit ←	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

main() function contains implicit call to system call exit(). Thus, when main thread finishes main() function, there exit() is executed and process is killed.
When main thread calls pthread_exit(), this function is non-returnable, means that main thread would not return to main() function anymore. This is why exit() system call would not be executed in this case, and the process (together with all rest of its threads) would continue.

ת'רדים ב-POSIX

הפרויקט ה-POSIXThreads הוא חבילת הננתמכת ברוב מערכות ה-UNIX והוא מגדירה למולא מ-60 קרייאות מערכת.

פונקציות מרכזיות ב-Threads

- **pthread_create:** יוצרת ת'רד חדש.
- **pthread_exit:** מסיימת את הת'רד הקורא.
- **pthread_join:** ממתינה לת'רד מסוים להסתיים.
- **pthread_yield:** משחררת את המעבד כדי שת'רד אחר יוכל לרוץ.
- **pthread_attr_init:** יוצרת ומתחילה מאפיין של ת'רד.

- `pthread_attr_destroy`: מסירה מאפיין של תред.

פרטים על 3 קריאות מערכת מרכזיות ב-Pthreads :

1. `pthread_create`:

קלט: מקבלת פרמטרים הקשורים לייצור הת'רד.

פלט: מחזירה את המזהה של הת'רד החדש שנוצר. היא דומה לקריאת המערכת `fork`, כאשר המזהה של הת'רד משחק תפקיד של PID ליזויו ת'רידים בקריאות אחרות.

2. `pthread_exit`:

קלט: מצב סיום או פרמטר ספציפי אחר.

פלט: הת'רד המתבצע נסגר ולא חוזר. הזיכרון המשויר לת'רד משוחרר.

3. `pthread_join`:

קלט: מזהה של הת'רד שעליו להמתין.

פלט: הת'רד הקורא ממתין עד שההת'רד המשויר מסתיים.

מאפיינים של Pthreads

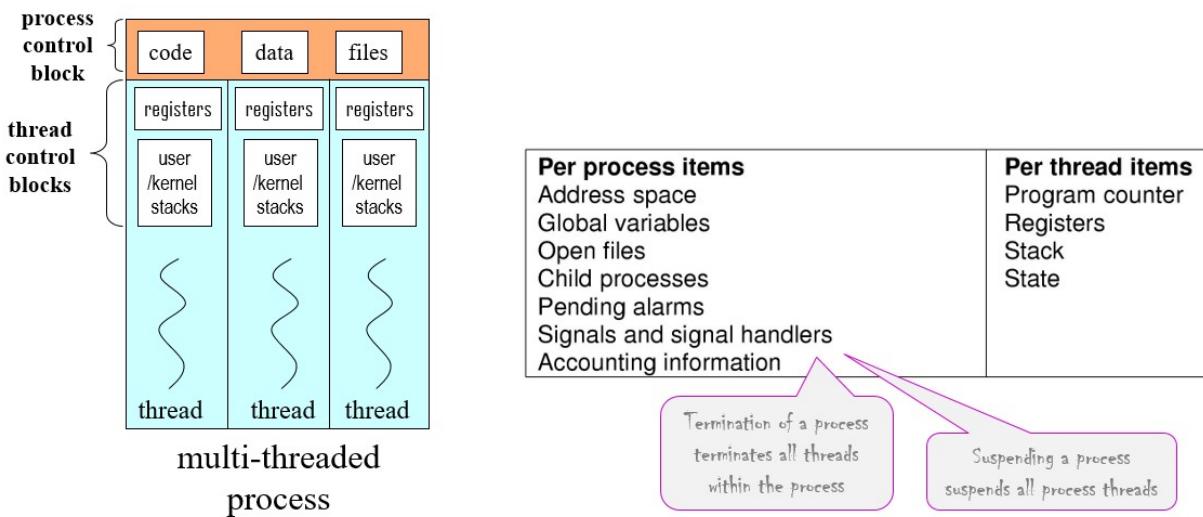
לכל `pthread` יש תכונות מסוימות, כולל מזהה, קבוצה של גיסטרים וקבוצה אחרת של תוכנות, אשר מאוחסנות במבנה המאפיינים. אלה כוללים את גודל הסטאק, פרמטרים לתזמן ועוד.

הקריאה ל-`main()` מכילה באופן רשמי קריאה לפונקציה המערכת `exit()`. כאשר הת'רד הראשי מסיים את הפונקציה `main()`, הקריאה `exit()` מתבצעת והתהליך נסגר. אם הת'רד הראשי מפעיל את `pthread_exit()`, קריאה זו איננה חוזרת, כלומר הת'רד הראשי לא יחזיר לפונקציה `main()`. לכן, במקרה זה לא תתבצע הפונקציה `exit()` והתהליך, יחד עם שאר הת'רידים שלו, ימשיך לroz.

הבדלים בין Java ל-Pthreads

בג'אווה, ה-JVM מופעלת מעל התוכנית, הדבר שחייב ליעול בקוד ב-C++. ה-JVM בג'אווה יכולה להחליט אם ת'רידים מסוימים יהיו אמיתיים או לא. ב-Pthreads, ישנו כלים שאינם קיימים בג'אווה, כמו `cond` ו-`mutex`.

PCB fields: process \leftrightarrow thread



מבנה תהליכי מרובה ת'רדים:

במערכת הפעלה, מערכת מרובה ת'רדים מאפשרת למספר ת'רדים לרוץ בו זמנית.

א. מה שימושת לכל הת'רדים: מסוימים כמו משתנים גלובליים, קבצים פתוחים, תהליכיBIN וSIGALRM.

ב. ייחודי לכל ת'רד: לכל ת'רד יש את ה-PC, הרגיסטרים שלו, מחסנית ומצב שלו – גם במרחב ה kernel וגם במרחב המשתמש.

כשמדובר במערכת בעלת מעבד אחד, הת'רדים מחליפים אחר אחר בהרצאה, וע"י זה נוצרת התהcosaה שכאילו הם רצים במקביל. בפועל, המעבד מחליף ביניהם בצורה מהירה, ומספק את ההגשה שהת'רדים רצים במקביל.

מצבי ת'רד:

כמו בתהליכי רגיל, ת'רד יכול להיות באחת ממספר המצבים: רץ, חסום, מוכן או הופסק. לדוגמה, כאשר ת'רד מבצע קריאת מערך למשך מהמקלדת, הוא יփוך למצב חסום עד שהקלט יוזן.

שאלה: האם יתכונו 4 ת'רדים אחד RUNNING, אחד BLOCKED ואחד Suspended?

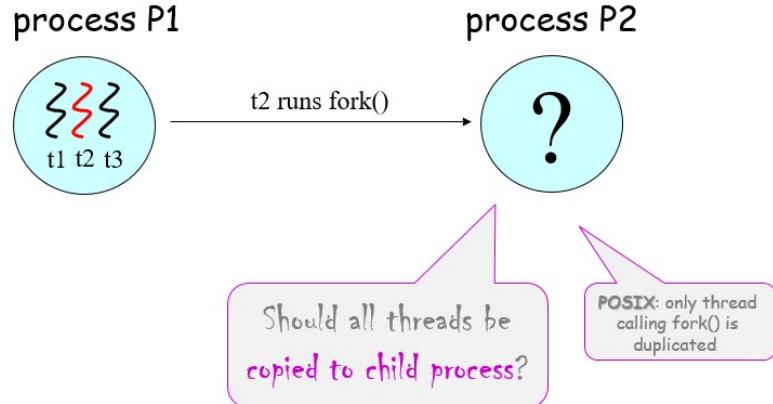
תשובה: לא הגיוני, שכן אם יש ת'רד במצב Suspended, כל הקוד שלו, במיוחד ה-PI, נמצא ברם.

הערות חשובות

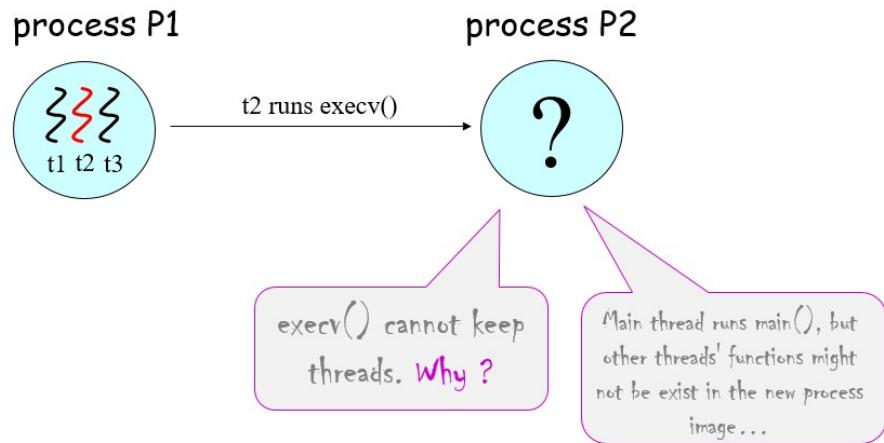
א. מטרת הת'רד היא לאפשר למספר ת'רדים לשתף פעולה במערכת הפעלה כדי להשיג מטרה מסוימת.

ב. מטרה לעבוד על מערכת הפעלה במתירה להפעלה נתקלת במנגנונים המונעים מתהליך לאנוב את תשומת הלב של המערכת.

Threads in Child Process



Threads in Child Process



ת'רדים בפעולת `fork` בתהיליך יلد

הת'רדים מעבר להיותם שימושיים, מעלים מספר בעיות אדולות. בעיקר, כאשר מדובר בפקודת `fork` של UNIX. אם לתהיליך החורה יש מספר ת'רדים, האם גם לתהיליך הילד צריך להיות אותם הת'רדים? ואם לא, התהיליך יכול לא לפעול כהלאה. אם הילד קיבל את כל הת'רדים, מה יקרה אם אחד הת'רדים בחורה היה נחסם על פקודה קריאה, לדוגמה, מהמקלדת? הבעיה אינה מסתירים מכך, ודרישה בחירה בחורה מצד מעוצבי המערכת הפעלה כדי להבין את התנהלות הת'רדים.

כאשר תהיליך, נניח P1, הכולל את הת'רדים t2, t1, וt3, מפעיל את הפקודה `fork()` באמצעות t2, נתקלים בבעיה מעניינת: האם כל הת'רדים בתהיליך החורה צריכים להתשכפל לתהיליך הילד, P2? בפרוטוקול POSIX החלטה היא כי רק הת'רד שיביצע את הפקודה `fork()` יתשכפל לתהיליך הילד.

از מה קורה כאשר ת'ירד, נניח 3^a, יתשכפל אך הוא ממתין לפעולות SO? איך אפשר ממש דבר כזה בתהיליך החדש? האוניות, זה בלתי אפשרי.

במהלך הפעולה exec(), אין אפשרות לשמר את הת'ירדים. הסיבה היא שכאשר התהיליך מחליף את תמונה הזיכרון שלו, הת'ירדים מרים פונקציות מהתמונה הקודמת. לאחר החלפת התמונה, הפונקציות אלו לא קיימות יותר, ולכן הת'ירדים אינם יכולים להמשיך לroz.

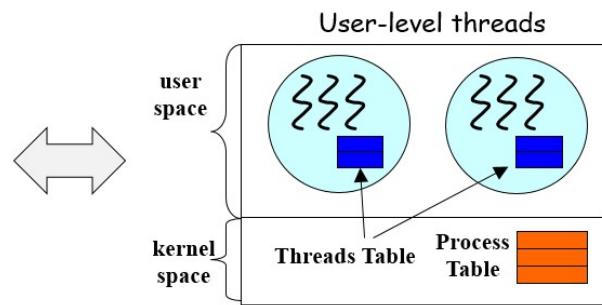
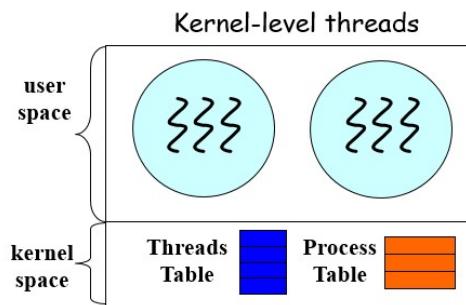
חשוב להבהיר כי יש המונח "הת'ירד הראשי" מתיחס לת'ירד אשר מרים את הפונקציה main(). זה לא אומר שהוא הת'ירד היחיד בתהיליך.

Kernel-level vs. User-level threads procs ?

- Only single CPU can be used
- All thread management is done by user
 - user must implement Thread Scheduler
 - cannot handle clock interrupts per thread
 - thread must give up CPU voluntary
- System call blocks the entire process
 - we may use asynchronous I/O instructions

cons ?

- Threads' scheduling is application-specific
 - may be more efficient for a given application
- Thread context switch happens in user mode
 - faster context switch



ת'ירדים במרחב הקרנלי מול ת'ירדים במרחב המשתמש:

הניהול של ת'ירדים יכול להתבצע בשני דרכים מרכזיים: במרחב המשתמש ובמרחב הקרנלי.

א. ת'ירדים במרחב המשתמש:

בשיטה זו, כל חבילה הת'ירדים נמצאת במרחב המשתמש, והקרנלי אינו מודע לקיומם. מבחיננו, הוא מנהל תכניות כאלו יש להן ת'ירד אחד בלבד. היתרון העיקרי הוא האפשרות למשתמש את הת'ירדים אלו במערכות הפעלה שאינן תומכות בת'ירדים.

דוגמאות: ת'ירדים בשפת Java, קורוטיניות ב-PPL.

חסרונות:

- א. קריאות מערכת חוסמות: אם ת'ירד עושה קריאה מהמקלדת, הקריאה יכולה לחסום את כל התהיליך.
- ב. בעיות עם page faults: הקרנלי אינו מודע לת'ירדים ויחסם את כל התהיליך במקרה של page fault.
- ג. אם ת'ירד מתחילה לroz, ת'ירדים אחרים לא יוכלו לroz עד שהת'ירד הראשון יouter מרצונו על זמן CPU.

ב. ת'ירדים במרחב הקרנלי:

בשיטת זו, הkernel מודע לכל הת'רדים ומנהל אותם באופן פעיל. הkernel מחזק בטבלת ת'רדים הכוללת את מצב הת'רד, הרגיסטרים שלו ומידע נוסף. כאשר ת'רד רצה ליצור או להרשות ת'רד אחר, הוא פונה לkernel.

דוגמא: הספרייה `pthread` שנראית SPL בклиינט שכתוב ב-C++.

יתרונות:

הkernel מסוגל לנוהל ת'רדים בצורה הרבה יותר יעילה, ובמקרה של page fault או חסימה אחרת, הוא יכול לבחור איזה ת'רד להריץ.

חסרונות:

- א. בעיה במימוש הפקודה `fork`: לא ברור אם התהילה החדש צריך להכיל את כל הת'רדים או רק חלקם.
- ב. ניהול סיג널ים: הסיגナルים מיועדים לת'רדים ולא לת'ליכים, וכך לאחר סיגナル מתתקבל, לא ברור איך לנוהל את התהילה.

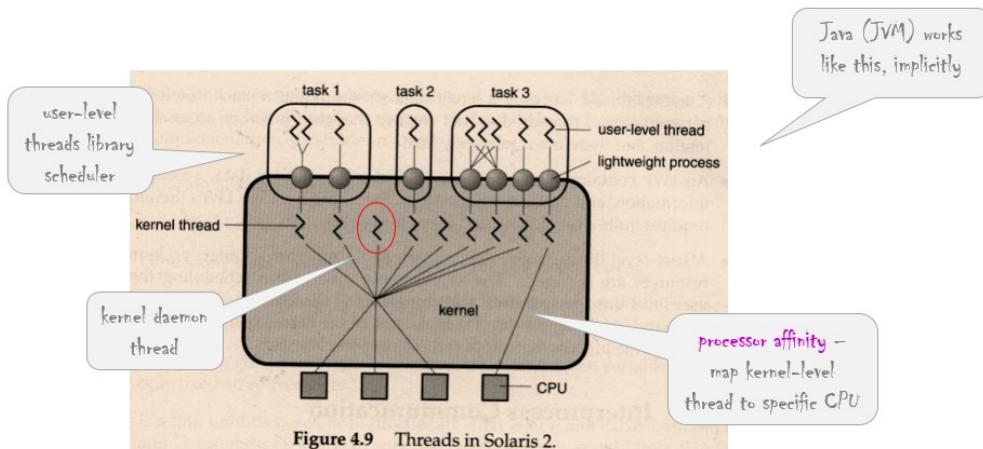
לסיכום:

חסרוןות: השימוש האפשרי בלבד, בעיות בניהול סיגナルים ובביצוע הפקודה `fork`.

יתרונות מפוקפקים: התוכנית בוחרת איך לזמן ת'ליכים, אך מערכת הפעלה יכולה לעשות זאת באופן יותריעיל.

Solaris OS: Combined Approach for Threads

- ❑ **User-level threads may be mapped onto kernel-level threads via LWP (light-weight process) object.**



מימוש היברידי:

אפשר שימוש בגמישות של ת'רדים במרחב המשתמש ובאי-תנות של ת'רדים במרחב kernel. הדבר נעשה על ידי ערבות ת'רדים במרחב המשתמש לת'רדים במרחב kernel.

במודל זה לתוכנת יש גמישות בקביעת מספר הת'רדים אשר יהיו בkernel ואשר יהיו במרחב המשתמש.

הkernel מודע רק לת'רדים בرمתו ולכן יתמן רק אותם. הת'רדים במרחב המשתמש נוצרים, נהרסים ומתחזנים כמו כל ת'רד מרחב המשתמש במנגנון הרגיל. כל ת'רד בرمת kernel הוא קבוצה של ת'רדים במרחב המשתמש.

יתרונות:

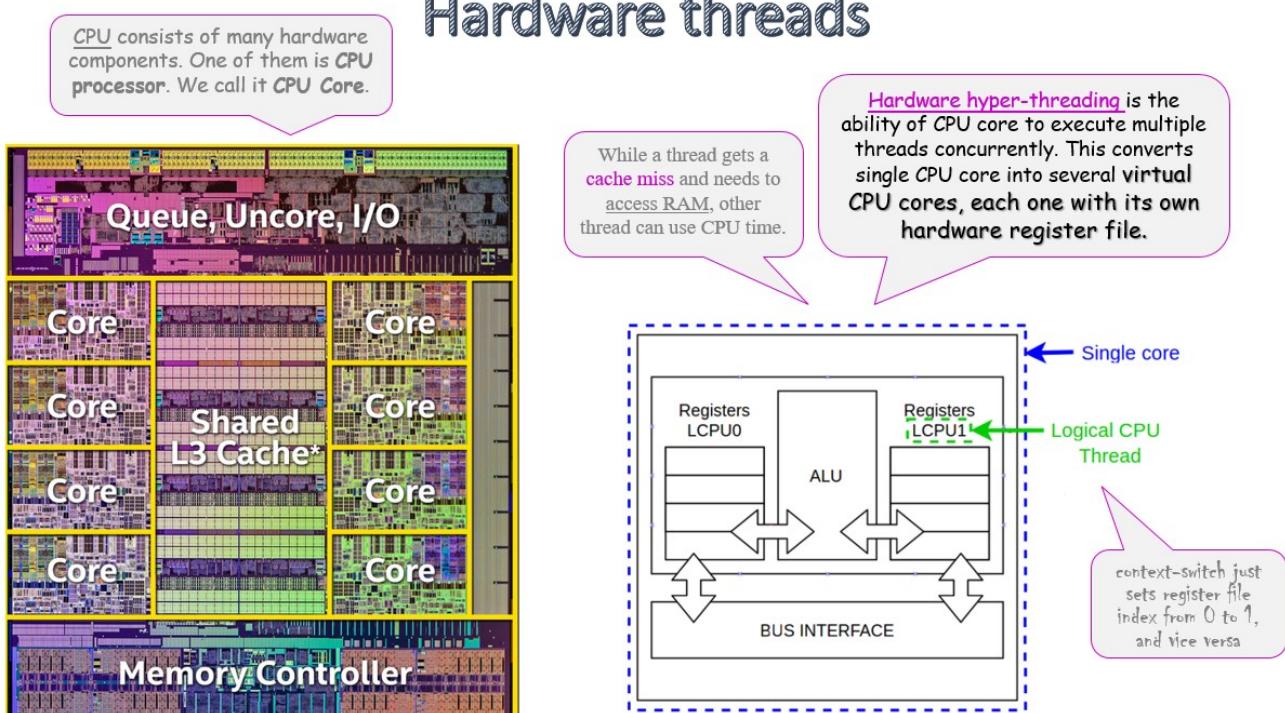
א. שימוש יעיל במשאבי המערכת.

ב. אמישות בהתאם מספר הThreads במרחב המשמש אל מול מספר הת'Redis' במרחב הkernel בהתבסס על עבודה ודרישות מערכת.

ג. אפשר יצירה, מחיקה וזמן של Threads של מרחב המשמש מבלי לעرب את הkernel. בכך להוריד את overhead.

תרשים של מערכת הפעלה מתחABL מזכירה את התרשים של גיאו: כאשר Thread בرمת הkernel שעומד זמן שהוא קיבל מתבצעים כמה קו-הוטינוט. יש גם kernel threads שלא עושים כלום ויש כאלו שMRIIZIM Thread של משתמש.

Hardware threads



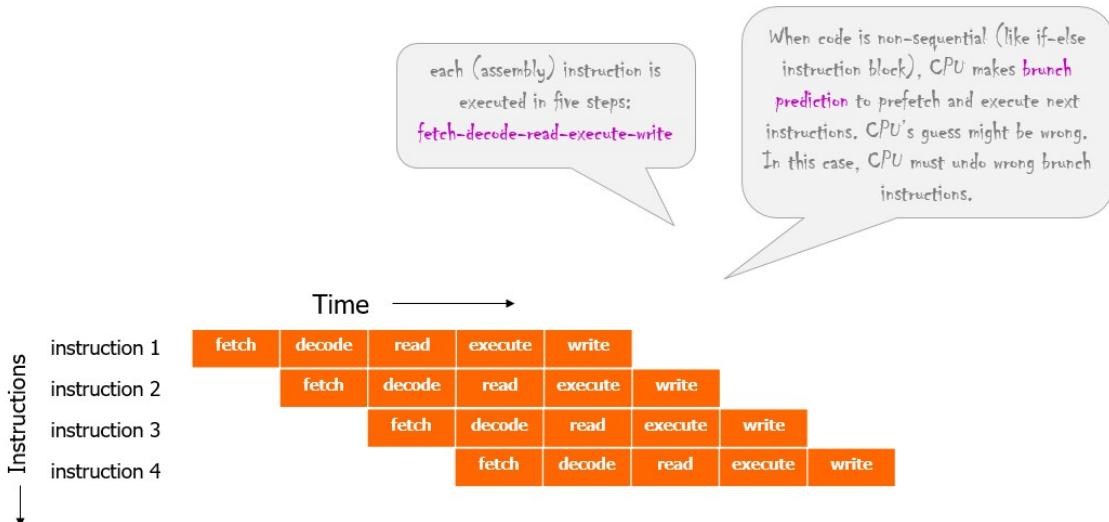
ת'Redis' בرمת החומרה:

המעבד (CPU) מכיל מספר רב של רכיבי חומרה. אחד הרכיבים החשובים בו הוא ליבת המעבד, שאנו מכנים אותה "ליבת CPU". במהלך פעולה המעבד, כאשר תחילך אחד נתקע וצריך לגשת ל זיכרון הראשי (RAM), תחילך אחר יוכל להשתמש בזמן המעבד.

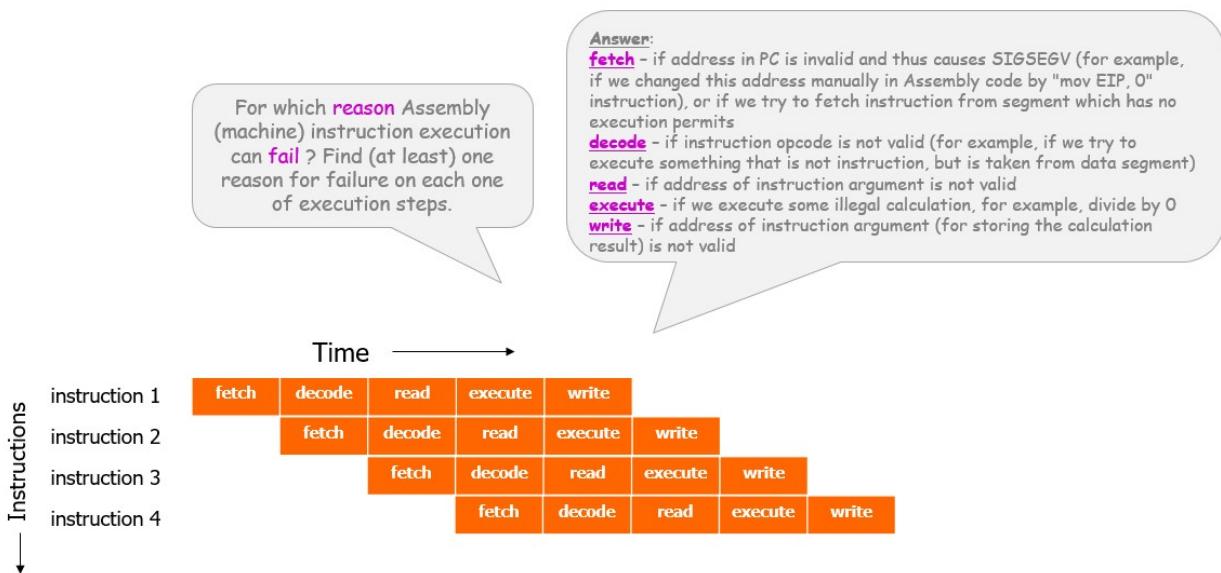
אחד הטכנולוגיות המתקדמות בתחום המעבדים היא תחילך ה"הייפר-ת'Redis' (Hyper-Threading). זו היכולת של ליבת המעבד לבצע מספר תהליכים בו זמנית. כך המעבד ממיר את ליבת CPU הבודדת למספר ליבות וירטואליות, כאשר לכל אחת מהן יש קובץ גיסטר חומרתי מיוחד. החלפת ההקשר בין התהליכים (-Context Switch) פשוטה - היא מעבירה את מספר הרגיסטר מ-0 ל-1, ולהיפך.

בנוסף, ישנים תהליכי אחרים המתקיימים בرمת החומרה. כאשר אנו מדברים על ליבות המעבד, כל ליבה יכולה להריץ תחילך מסוים. ישנים זכרונות מטמון (Cache) בرمות L1 ו-L2 הם פרטיים לכל ליבה, בעוד L3 הוא משותף. אם נגדיל את הליבה, נוכל לראות יחידה ארitmética ושני סטים של גיסטרים. הסיבה לכך היא שלhiba אחת מסוגלת להריץ שני תהליכים בו זמנית, מה שמאפשר לליבה לבצע פונקציה אחרת בזמן שהוא מגשת ל זיכרון הראשי (RAM).

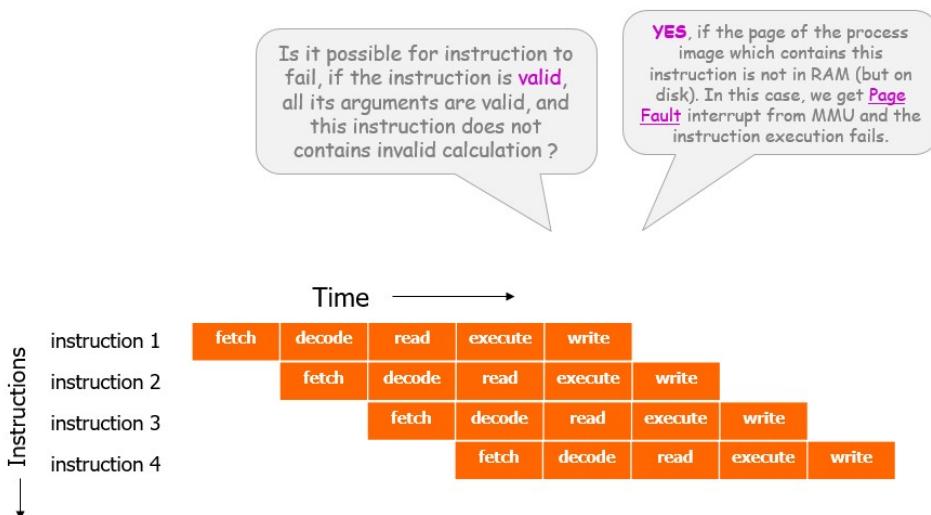
Pipeline : Instruction-Level Parallelism (ILP)



Pipeline : Instruction-Level Parallelism (ILP)



Pipeline : Instruction-Level Parallelism (ILP)



הבנייה ה-Pipeline ומקביליות ההוראות (ILP):

בעזרת ה-Pipeline, כל הוראה (ב-Assembly) מתבצעת בחמישה שלבים:

1. Fetch
2. Decode
3. Read
4. Execute
5. Write

כאשר הקוד אינו סידרתי (כמו בлок if-else), ה-CPU עושה חיזוי להבאת ההוראות הבאות והרצתן. יתכן והחיזוי יתברר כשגוי, ובמקרה זה, ה-CPU תצטרכ לבטל את ההוראות של החיזוי השגוי.

מתי ההוראה ב-Assembly עלולה להיכשל?

- Fetch - לדוגמה (SIGSEGV) אם הוראה לאינה חוקית, דבר שיגרום לשגיאת PC-כאשר הכתובת ב-PC.
- Decode - אם קוד ההוראה אינו חוקי.
- Read - אם כתובות הארגומנט של ההוראה אינה חוקית.
- Execute - במקרה של חישוב בłęתי חוקי, כמו חלוקה באפס.
- Write - אם כתובות הארגומנט לשימירת התוצאה אינה חוקית.

אפשר להיתקל בטעיה בהוראה אם כאשר ההוראה עצמה והארוגומנטים שלה תקפים, במקרה שבו הדף שבו ההוראה מאוחסנת אינו נמצא ב-RAM אלא בדיסק. במקרה זה, תוחזר שגיאת "Page Fault" מה-MMU (אנו נדונן על כך בפרק על זיכרון).

הגברת הביצועים תוך ניצול מערך ה-Pipeline:

ברגע שהמעבד מבצע את השלב הראשון של הוראה, הוא ממשיר מיד להוראה הבאה. היתרון בכך הוא שיכל שהתקנית מתבצעת בצורה יותר סידרתיות, היא אם תרצו בצורה מהירה יותר. לדוגמה, אם השורה הראשונה היא תנאי של if, והשורות הבאות הן ההוראות שמתבצעות כאשר התנאי מתקיים, אם התנאי הוא שקר - יש להעתם מכל ההוראות הבאות.

Process Management : outline

- Process states and structures
- Process management
- Inter-process communication
- Signals
- Threads
- Specific implementations

Linux Processes and Threads – sharing options

sharingFlags is a binary array
that specifies which parts of
process to share

pid = **clone(function, stackPtr, sharingFlags, arg);**

“...creates a new process with possibly shared resources...”

VM = Virtual
Memory

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table

bits in the sharingFlags bitmap

When all flags are set (i.e., are 1), we
get a new thread. When all flags are
cleared (i.e., are 0) we get new process.

יצירת תהליכי ות'רידים בLINNOKS

בלינוקס, ישנו מספר דרכים ליצור תהליכי ות'רידים. אפשר להשתמש ב- `pthread_create` ליצירת ת'רידים. ב- `fork` ליצירת תהליכי, או באמצעות פונקציית `clone`, שמטשטשת את ההבדלים בין השניים.

פונקציית `clone` מציעה גמישות רבה בנוגע לשיתוף משאים בין התהליכים והת'רידים. היא מאפשרת להחליט באופן דינامي אילו משאים ישוטפו ואילו לא, באמצעות הפרמטר `sharingFlags`. כאשר כל הדגמים מוגדרים

ל-1, הפקנץיה תיצור ת'רד חדש באותו התהילר. אם כל הדגלים הם 0, היא תיצור תהילר חדש.

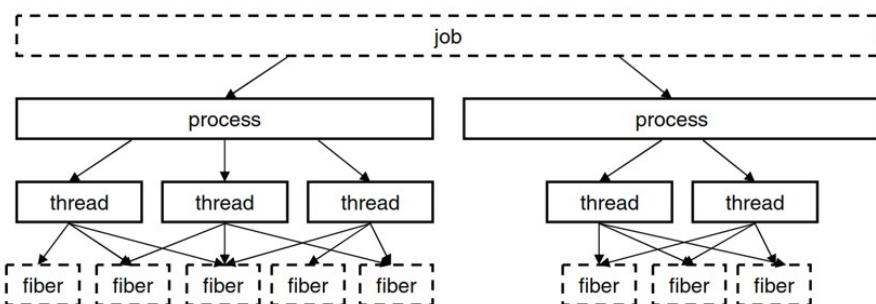
אם הדגל CLONE_VM מוגדר ל-1, הפקנץיה תיצור ת'רד חדש; אם הוא 0, היא תיצור תהילר חדש.

בניגוד לגישה הקלאסית של יוניקס, לינוקס מבחינה בין מזהה לתהילר (PID) לבין מזהה למשימה (TID). זאת אומرت שבלינוקס כל ת'רד יכול לקבל מזהה משלי, בעוד שביונייס כל הת'רדים בתהילר משתפים את אותו המזהה.

Windows – Processes and Threads

Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

Shared limits: total number of threads for all the processes together, the highest possible (user) priority for these processes, total RAM usage, and so on.



וינדוס - תהילכים ות'רדים

בוינדוס ישנו מספר רכיבים שאנו יכולים ליצור ולנהל: תהילכים, ת'רדים, ועובדות. כאשר יוצרים עבודה, אפשר להגדיר לה גבולות מסוימים כמו הכמות המרבית של ת'רדים שאפשר להפעיל, הגבול העליון של זיכרון RAM שהעבודה יכולה להשתמש בו, וכן הגבול העליון לעדיפות התהילר.

כמו כן, בוינדוס יש אפשרות ליצור "ת'רדים קלים" אשר מתבצעים במרחב המשתמש, ונקראים "FIBER".

בצורה ארכיטקטורתית, ההיררכיה היא כז':

- JOB (עובדת)
- PROCESS (טהילר)
- THREAD (ת'רד)
- FIBER (ת'רד קל)

כאשר מייצרים ומנהלים את המשאבים הללו, יש לזכור בחשבון את הגבולות המשותפים והמגבילות המוגבלות עבור כל התהילכים והת'רדים.



© 2022 מתקנים הפעלה

Operating Systems

Lecture 2 – Process Management

Dr. Marina Kogan-Sadetsky