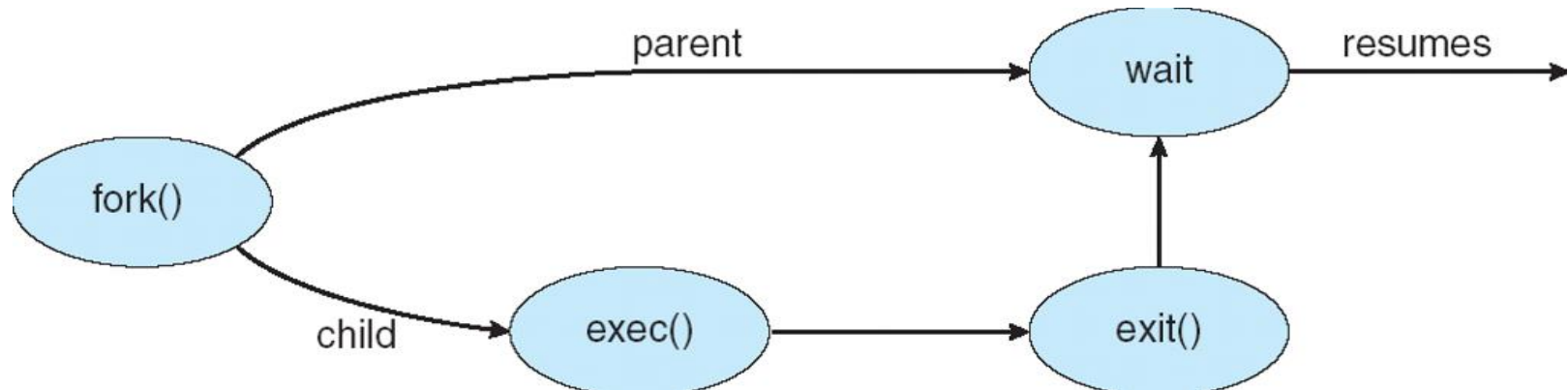


תרגול 2

קריאות מערכת לעבודה עם תהליכים
ניהול תהליכים בגרעין לינוקס
תורי תהליכים

TL;DR

- תכנית היא אוסף פקודות; תהליך הוא ביצוע של אותן פקודות.
- נלמד איך קוד משתמש יכול ליצור תהליכים חדשים, לברר מה מצבם, ולהמתין לסיום שלהם.
- באמצעות קריאות המערכת: **fork**, **execv**, **exit**, **wait** (ועוד כמה).
- הממשק לא אינטואיטיבי במבט ראשון.



- נלמד איך הגרעין מממש את קריאות המערכת הללו.

קריאות מערכת לעבודה עם תהליכים

מהו תהליך?

- תהליך (**process**) הוא ביצוע סדרתי של תכנית (**program**).
• תהליך = מופע (instance) של ביצוע תכנית.
- מערכת ההפעלה נותנת לכל תהליך **אשליה** שהוא לבד במערכת כדי להקל על פיתוח אפליקציות וכדי לספק לתהליכים הגנה זה מזה.
- אבל תהליכים יכולים גם לתקשר ביניהם – נלמד בהמשך הקורס.
- מספר תהליכים רצים "בו-זמנית" על המעבד: מערכת ההפעלה מחליפה בין התהליכים במהירות ויוצרת אשליה שהם רצים יחד.
• בהמשך הקורס נלמד איך לינוקס מממשת את ההחלפה בין התהליכים.
- כל תהליך **צורך משאבים**, למשל: זמן מעבד, זיכרון, ...
• בהמשך הקורס נלמד איך לינוקס מחלקת את זמן המעבד בין התהליכים.

תהליכים בסביבת Windows

מנהל המשימות

קובץ אפשרויות תצוגה

תהליכים ביצועים היסטוריית אפליקציות אתחול משתמשים פרטים שירותים

שם	PID	CPU	זיכרון	דיסק	רשת
אפליקציות (5)					
(3) Firefox		1.3%	294.3 MB	0.1 MB לשניה	0 Mbps
Microsoft PowerPoint	7096	0%	75.4 MB	0 MB לשניה	0 Mbps
Microsoft Word	12952	0%	44.5 MB	0 MB לשניה	0 Mbps
מנהל המשימות	12872	0.2%	24.5 MB	0 MB לשניה	0 Mbps
סייר Windows	13988	0.3%	36.8 MB	0.1 MB לשניה	0 Mbps
תהליכים ברקע (78)					
...32) Adobe Acrobat Update Service	4604	0%	1.0 MB	0 MB לשניה	0 Mbps
Antimalware Service Executable	11284	0.3%	106.8 MB	0.1 MB לשניה	0 Mbps
Application Frame Host	9768	0%	3.9 MB	0 MB לשניה	0 Mbps

סיים משימה

פחות פרטים

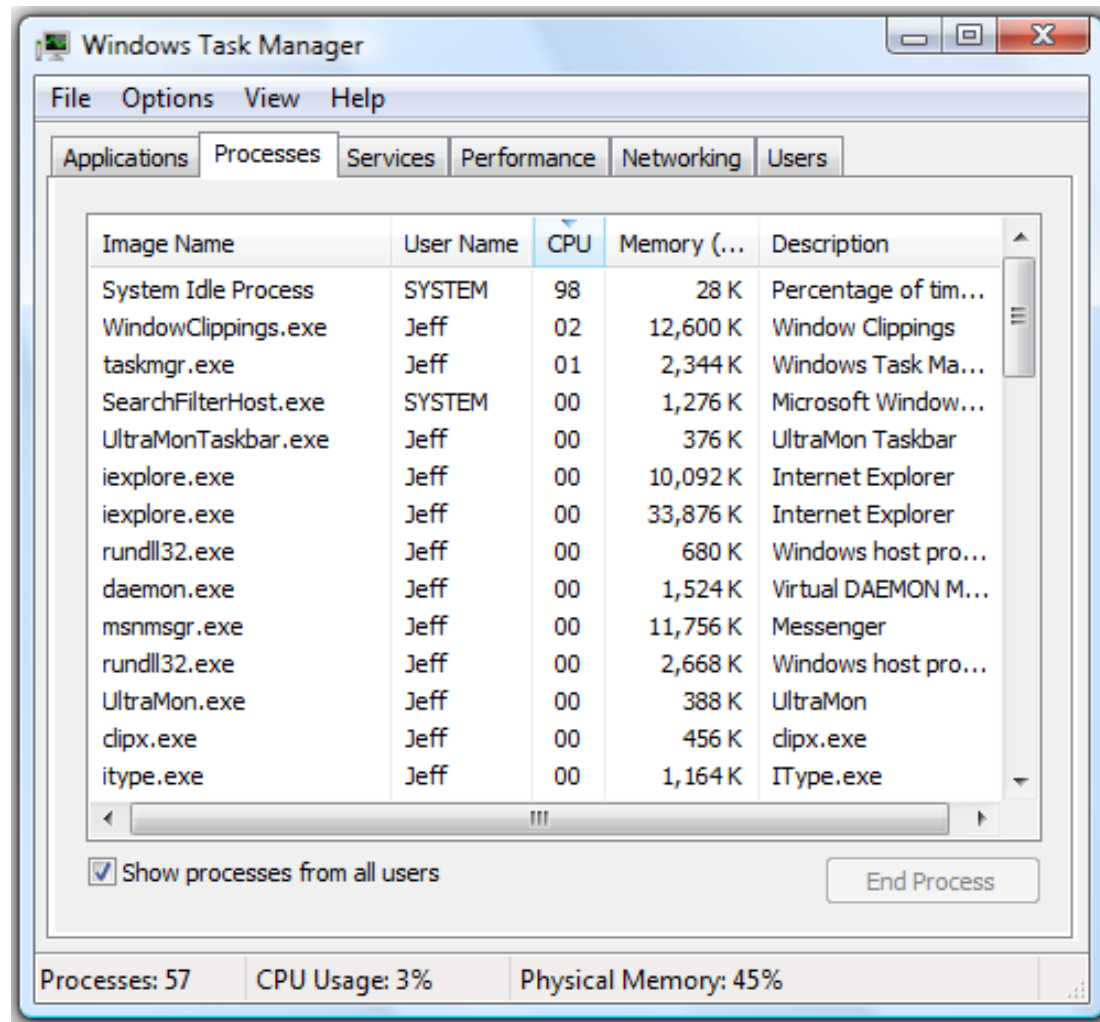
תהליכים בלינוקס

- לכל תהליך בלינוקס יש מזהה הקרוי **PID** – Process Identifier.
- מספר שלם בן 32 ביט, ייחודי לתהליך.
- ברוב מערכות לינוקס משתמשים רק ב-15 הביטים התחתונים, ולכן ניתן ליצור עד 32K תהליכים. מנהל המערכת יכול להגדיר מספר גבוה יותר של תהליכים.
- שימו לב: ערכי ה-pid ממוחזרים מתהליכים שסיימו לתהליכים חדשים.
- עם עליית המערכת, הגרעין יוצר את התהליך **idle** שמספרו **pid=0**.
- נקרא לריצה כאשר אין תהליכים מוכנים לרוץ.

למה זה כדאי?

מכונה **hlt**, המכניסה את המעבד למצב שינה.
- התהליך **idle** יוצר את התהליך **init** שמספרו **pid=1**.
- התהליך **init** ייצור את כל שאר התהליכים.

Windows בסביבת Idle



קריאת המערכת `fork()`

```
pid_t fork ();
```

- פעולה: מעתיקה את תהליך האב לתהליך הבן וחוזרת בשני התהליכים.
- קוד זהה (ומיקום בקוד).
- זיכרון זהה (משתנים וערכיהם, גם במחסנית וגם בערימה).
- סביבה זהה (קבצים פתוחים, ספריית עבודה נוכחית).
- אבל, תהליך הבן הוא תהליך נפרד מתהליך האב, לכן יש לו **PID משלו**.
- פרמטרים: אין.
- ערך מוחזר:
 - במקרה של כישלון: -1 – לאב (אין בן).
 - במקרה של הצלחה: לבן מוחזר 0 ולאב מוחזר ה-**pid של הבן**.

איך נוכל להבדיל בין אב לבן אם ה-`pid` של הבן יוצא במקרה 0?

לפני fork()

parent

```
int main() {  
    int x = 0;  
    → pid_t p = fork();  
    if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

אחרי fork()

parent

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    → if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

son

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    → if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

קוד האב וקוד הבן מסתעפים

parent

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    if (p == 0) {  
        x = 1;  
    } else {  
        → x = 2;  
    }  
}
```

son

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    if (p == 0) {  
        → x = 1;  
    } else {  
        x = 2;  
    }  
}
```

מה יהיה ערכו של x
בסופו של דבר?
1 או 2?

שכפול מרחב הזיכרון ע"י fork()

- לאחר פעולת fork() מוצלחת, אמנם יש לאב ולבן את אותם משתנים בזיכרון, אך **בעותקים נפרדים**.
- כלומר, שינוי ערכי המשתנים אצל האב לא ייראה אצל הבן, וההיפך.

הדפסה לא מתואמת למסך

- שאלה: מה מדפיס הקוד הבא
- אם fork נכשלת? ואם היא מצליחה?

hello

- ואם fork() הצליחה?

- יש הרבה תשובות אפשריות, למשל:

hellohello

hheelloollo

helhellolo

```
int main() {
    fork();
    printf("hello");
    return 0;
}
```

- הסיבה: שני התהליכים ניגשים בצורה לא מתואמת למשאב משותף – המסך.

קריאת המערכת wait()

```
pid_t wait(int *wstatus);
```

- פעולה: ממתינה עד אשר אחד מתהליכי הבן יסיים.
- פרמטרים:
 - wstatus – מצביע למשתנה בו יאוחסנו פרטים על תהליך הבן שהסתיים.
 - למשל, wstatus יכול את ערך הסיום של הבן (הערך שהעביר כארגומנט ל-`exit()`). ערך הסיום מופיע בבית השני מתוך ארבעת בתי ה-wstatus. כדי לחלץ אותו יש לנצל את המאקרו `WEXITSTATUS(*wstatus)`, המחזיר `(*wstatus >> 8) & 0xff`.
 - במידה ולא מעוניינים בסטטוס הבן שסיים, אפשר להעביר `NULL`.
- ערך מוחזר:
 - אם אין בנים או שכל הבנים כבר סיימו ובוצע להם `wait()` מבינים את החישוב? הערך -1.
 - אם יש בנים שסיימו ועדיין לא בוצע עבורם `wait()` (כלומר הם במצב **zombie** – יפורט בשקופיות הבאות) – יוחזר מיד ה-pid של אחד הבנים הנ"ל.
 - אחרת – המתנה עד שבן כלשהו יסיים.

מבינים את החישוב?

איך תהליך אב יכול לחכות
לסיום כל תהליכי הבן?

הדפסה מתואמת למסך

- שימוש ב-`wait()` יכול לפתור את הבעיה שראינו קודם כאשר מדפיסים למסך במקביל משני תהליכים:

```
int main() {  
    pid_t p = fork();  
    if (p > 0) {  
        // parent waits for child  
        wait(NULL);  
    }  
    printf("hello");  
    return 0;  
}
```

קריאת המערכת `waitpid()`

```
pid_t waitpid(pid_t pid, int *wstatus,  
               int options);
```

- פעולה: המתנה לסיום בן ספציפי שמספרו `pid`.
- `wait()`, `waitpid()` הן קריאות מערכת חוסמות.
- כלומר חוסמות את התקדמות התהליך עד להתרחשות תנאי מסוים.
- באנגלית: **blocking system calls**.
- הארגומנט `options` מאפשר לשנות את ההתנהגות של `waitpid()` לקריאת מערכת לא חוסמת.
- אם `options==WNOHANG` קריאת המערכת תחזור מיד, כאשר ערך חזרה 0 משמעותו שאף תהליך בן עוד לא סיים, ואילו ערך חזרה חיובי הוא ה-`pid` של תהליך בן שסיים ונמצא עדיין במצב `zombie`.

קריאת המערכת `exit()`

```
void exit(int status);
```

- פעולה: מסיימת את ביצוע התהליך הקורא ומשחררת את כל המשאבים שברשותו. התהליך עובר למצב **zombie** עד שתהליך האב יבקש לבדוק את סיומו ואז יפונה לחלוטין.

מה המטרה של מצב זה?

- פרמטרים:

- `status` – ערך סיום המוחזר לאב אם יבדוק את סיום התהליך.
- בפועל ניתן להעביר להורה רק 8 ביטים בתור ערך סיום, ולכן קריאת המערכת תעביר `(status & 0xff)`.
- ערך מוחזר: הקריאה אינה חוזרת.
- לפי ה-man, קריאת המערכת `exit` לא יכולה להיכשל.

קריאת המערכת `exit()`

- שאלה: למה בכלל לקרוא ל-`exit(status)`, אם אפשר פשוט לרשום `return status` בסוף פונקציית ה-`main`?
- תשובה: `main` היא לא באמת הפונקציה הראשית של התכנית...

- `main()` נקראת ע"י `__libc_start_main()` שאוספת את ערך החזרה של `main()` וקוראת ל-`exit()`.

```
int __libc_start_main(...) {  
    .....  
    exit(main(...));  
}
```

- מסקנה: הפונקציה `exit` תמיד נקראת לסיום סטנדרטי של התוכנית.

סיום תהליכים

- כדי לאפשר לאב לקבל מידע על סיום הבן, לאחר שתהליך מסיים את פעולתו הוא עובר למצב מיוחד – **zombie** – שבו התהליך קיים כרשומת נתונים בלבד ללא שום ביצוע משימה.
- הרשומה נמחקת לאחר שהאב קיבל את המידע על סיום הבן באמצעות `wait()`.
- שאלה: מה קורה לתהליך "יתום" (orphan), כלומר תהליך שסיים לאחר שאביו כבר סיים בלי לקרוא ל-`wait()`?
- התהליך הופך להיות בן של `init`.
- התהליך `init` ממשיך להתקיים לאורך כל פעולת המערכת.
- אחד מתפקידיו העיקריים – המתנה לכל בניו כדי לפנות את נתונייהם לאחר סיומם.

קריאת המערכת `execv()`

```
int execv(const char *filename,  
          char *const argv[]);
```

• פעולה: טוענת תכנית חדשה לביצוע במקום התהליך הקורא.

• פרמטרים:

• `filename` – מסלול אל הקובץ המכיל את התכנית לטעינה.

• `argv` – מערך מצביעים למחרוזות המכיל את הפרמטרים עבור התכנית.
האיבר הראשון מקיים `argv[0] == filename`, דהיינו מכיל את שם קובץ התכנית. האיבר שאחרי הפרמטר האחרון מכיל `NULL`.

• ערך מוחזר:

• במקרה של כישלון: -1.

למה צריך `NULL`?

• במקרה של הצלחה: הקריאה אינה חוזרת. איזורי הזיכרון, קהילה, מהתחלה, ... של התהליך מאותחלים עבור התכנית החדשה שמתחילה להתבצע מההתחלה.

קריאת המערכת `execv()`

• מה ידפיס הקוד הבא?

```
int main() {  
    char *argv[] = {"date", NULL};  
    execv("/bin/date", argv);  
    printf("hello");  
    return 0;  
}
```

• התשובה:

• אם `execv()` מצליחה: את התאריך והשעה.

• אם `execv()` נכשלת: hello

קריאות המערכת getpid(), getppid()

```
pid_t getpid();
```

- קריאת מערכת המחזירה לתהליך הקורא את ה-pid של עצמו.

```
pid_t getppid();
```

- קריאת מערכת המחזירה את ה-PID של תהליך האב של התהליך הקורא.

- שאלה: מה המשמעות של `getppid() == 1` עבור תהליך משתמש טיפוס?

- תשובה: תהליך האב הוא `init`. קורה למשל אם תהליך הבן יתום.

סיכום: עבודה עם תהליכים בלינוקס

- תהליך חדש יכול להיווצר אך ורק ע"י העתקה של תהליך קיים, באמצעות קריאת המערכת `fork()`.
- התהליך המקורי נקרא תהליך אב (או הורה), התהליך החדש נקרא תהליך בן.
- העותק של תהליך הבן זהה לגמרי פרט למזהה התהליך (PID).
- תהליך אב יכול ליצור יותר מתהליך בן אחד.
- לאחר היווצרו, תהליך הבן יכול לבצע משימה שונה מאביו על-ידי הסתעפות בקוד התכנית בהתאם לערך החזרה של `fork()`.
- תהליך הבן יכול לטעון תכנית חדשה לביצוע על-ידי קריאת המערכת `execv()`.
- תהליך אב יכול לבדוק או להמתין לסיום תהליך בן שלו על-ידי קריאת המערכת `wait()`.
- אב יכול לבדוק סיום של בנים שלו, אך לא של "נכדים", "אחים" וכדומה.

דוגמת קוד מסכמת

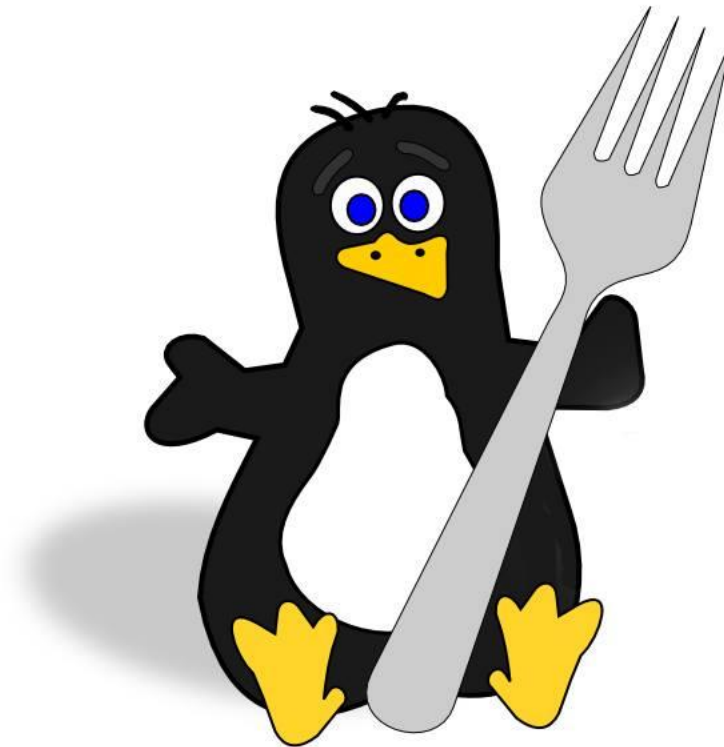
פלט לדוגמה:

```
pid = 8919
child pid = 8920
Sun Oct 29
00:31:32 IDT 2017
parent pid = 8919
```

```
printf("pid = %d\n", getpid());

pid_t pid = fork();
if (pid == 0) {
    printf("child pid = %d\n", getpid());
    char* args[] = {"/bin/date", NULL};
    execv(args[0], args);
    printf("This should not be printed\n");
} else {
    wait(NULL);
    printf("parent pid = %d\n", getpid());
}
```


הפסקה



i don't give a fork

אתחול תהליכים בלינוקס

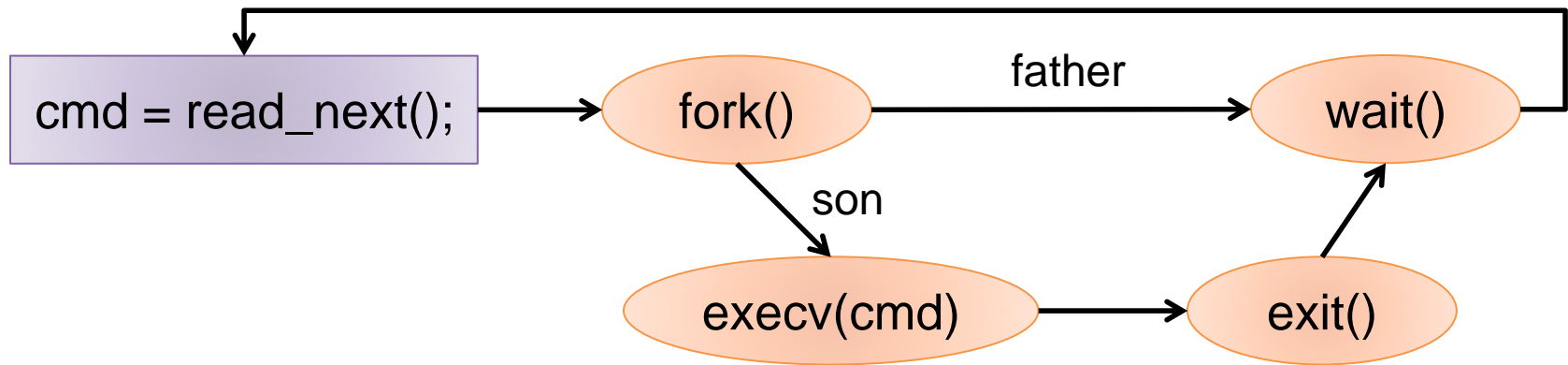
- משתמשים מתחברים לעבודה בלינוקס דרך מסופים (terminal).
- מסוף = מסך + מקלדת (מקומי או מרוחק).
- התהליך `init` יוצר תהליך בן עבור כל מסוף, אשר טוען ומבצע את המשימות הבאות לפי הסדר:
 1. איתחול של המסוף.
 2. התחברות של המשתמש עם שם משתמש וסיסמא באמצעות תכנית `login`.
 3. אם אושרה כניסת המשתמש: קריאה לתוכנית `shell` (כמו `tcsh` או `bash`) המאפשרת למשתמש להעביר פקודות למערכת ההפעלה.

דוגמה לשימוש בתהליכים – shell

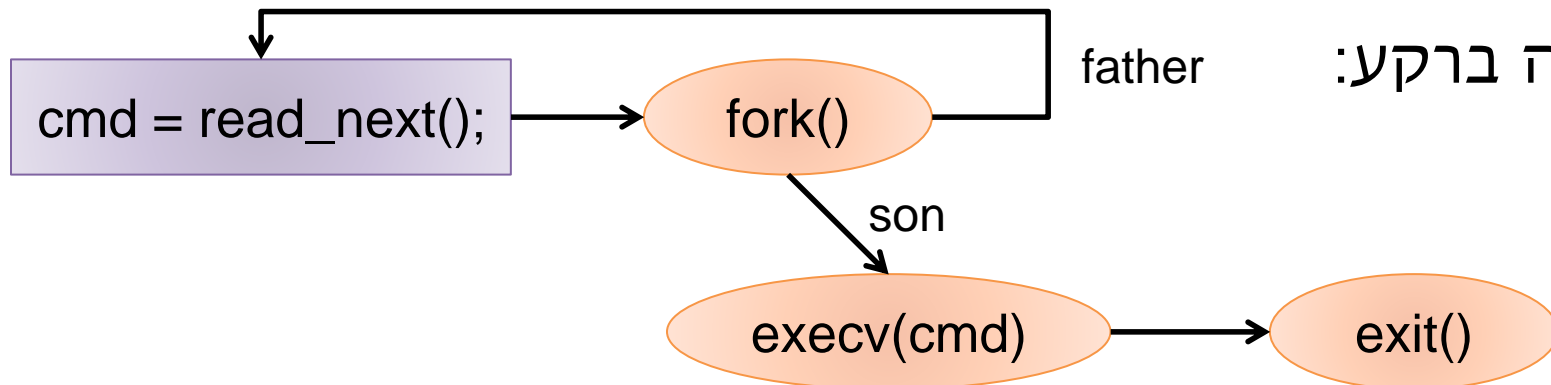
- ממשק שורת פקודה (command line).
- ייעוד עיקרי: לקבל פקודות ולבצע אותן באופן סדרתי.
- ה-shell מייצר תהליך בן עבור כל פקודה על-מנת לבצע אותה.
- כל פקודה ניתן להריץ בחזית (foreground) או ברקע (background).
- הרצה בחזית: האב (shell) ממתין לסיום הבן לפני קריאת הפקודה הבאה.
- הרצה ברקע: האב (shell) עובר מיד לקריאת הפקודה הבאה.
- ייעוד נוסף: להציג קבצים ותיקיות על-מנת לסייר במערכת.
- דוגמה חיה:
- https://www.tutorialspoint.com/unix_terminal_online.php

אופן פעולת shell בלינוקס

• הרצה בחזית:



• הרצה ברקע:





שאלה ממבחן

1. נתונות שתי תוכניות:

A

```
int main() {
    if (fork() > 0) {
        wait(NULL);
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

B

```
int main() {
    if (fork() > 0) {
        // do nothing
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

- מריצים את תוכנית A בלולאה בעזרת הפקודה הבאה ב-shell :
`for i in {1..N}; do ./prog; done`
- שאלה: כמה תהליכים לכל היותר יכולים להתקיים במערכת ברגע כלשהו?
- ללא התחשבות בתהליך ה-shell עצמו או תהליכים אחרים שאינם נתונים בשאלה.

א. 1. $2N$ ב. 2. $N+2$ ג. 3. $N+1$ ד. 4. N

ה. 5. 2

ו. 6. 1

- ומה התשובה אם מריצים את תוכנית B בלולאה?

2. נתונות אותן שתי תוכניות:

A

```
int main() {
    if (fork() > 0) {
        wait(NULL);
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

B

```
int main() {
    if (fork() > 0) {
        // do nothing
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

• כעת מריצים כל איטרציה של הכולאה ברקע :

`for i in {1..N}; do ./prog & done`

• שאלה: כמה תהליכים לכל היותר יכולים להתקיים במערכת ברגע כלשהו?

• ללא התחשבות בתהליך ה-shell עצמו או תהליכים אחרים שאינם נתונים בשאלה.

א. 1

ב. 2

ג. N

ד. N+1

ה. N+2

ו. 2N

• ומה התשובה אם מריצים את תוכנית B בלולאה?

ניהול תהליכים בגרעין לינוקס

מבני הנתונים לניהול תהליכים

- גרעין לינוקס מממש את קריאות המערכת שראינו (fork, wait, ...) באמצעות מבני הנתונים הבאים:
 1. **מתאר התהליך** (Process Descriptor).
 - במערכות הפעלה אחרות נקרא **PCB** = Process Control Block.
 - שומר בתוכו גם קשרי משפחה.
 2. **רשימת התהליכים** (Process list).
 3. **טבלת ערבול** **PID → PCB**.
 4. **Run Queue** – ("הטווח הקצר").
 5. **Waiting Queue** – ("הטווח הבינוני / ארוך").

מתאר התהליך


• לכל תהליך בלינוקס קיים בגרעין מתאר תהליך (`process` descriptor), שהוא אובייקט מטיפוס `task_struct` המכיל את:

- מזהה התהליך (PID).
- מצב הריצה של התהליך.
- עדיפות התהליך.
- מצביעים למתאר תהליך האב ו"קרובי משפחה" נוספים.
- מצביע לטבלת אזורי הזיכרון של התהליך.
- מצביע לטבלת הקבצים הפתוחים של התהליך.
- מצביעים למתארי תהליכים נוספים (רשימה מקושרת).
- מסוף איתו התהליך מתקשר.
- ועוד...

מימוש קריאת המערכת getpid()

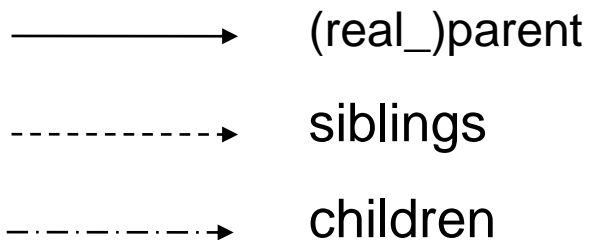
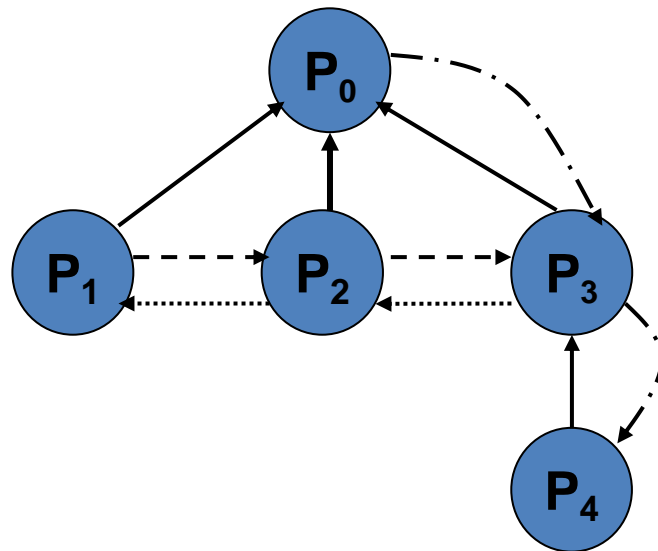
• מוגדרת בקובץ kernel/timer.c .

```
long sys_getpid(void) {  
    return current->pid;  
}
```



current הוא מצביע
למתאר התהליך הנוכחי.

ניהול קשרי משפחה בגרעין



- "קשרי המשפחה" בין תהליכים נשמרים באמצעות מצביעים הנשמרים ב-PCB.

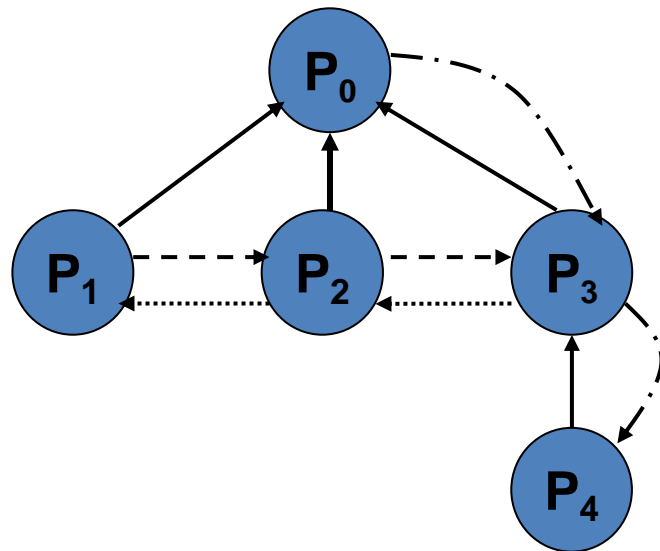
1. `real_parent`:

מצביע לאב המקורי.

2. `parent`: מצביע לאב בפועל.

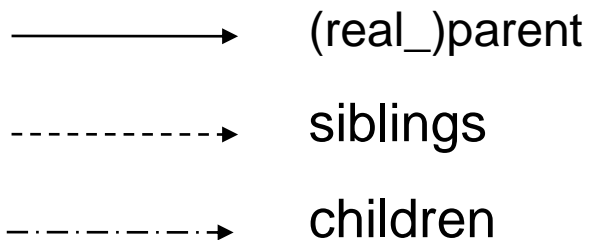
- האב בפועל שונה מהאב המקורי כאשר התהליך נמצא בריצה מבוקרת, למשל תחת `debugger`.
- כך תהליך יכול לאתר את אביו, למשל עבור קריאת `getppid()`.

ניהול קשרי משפחה בגרעין



3. children: מצביע לרשימה
מקושרת של בנים.

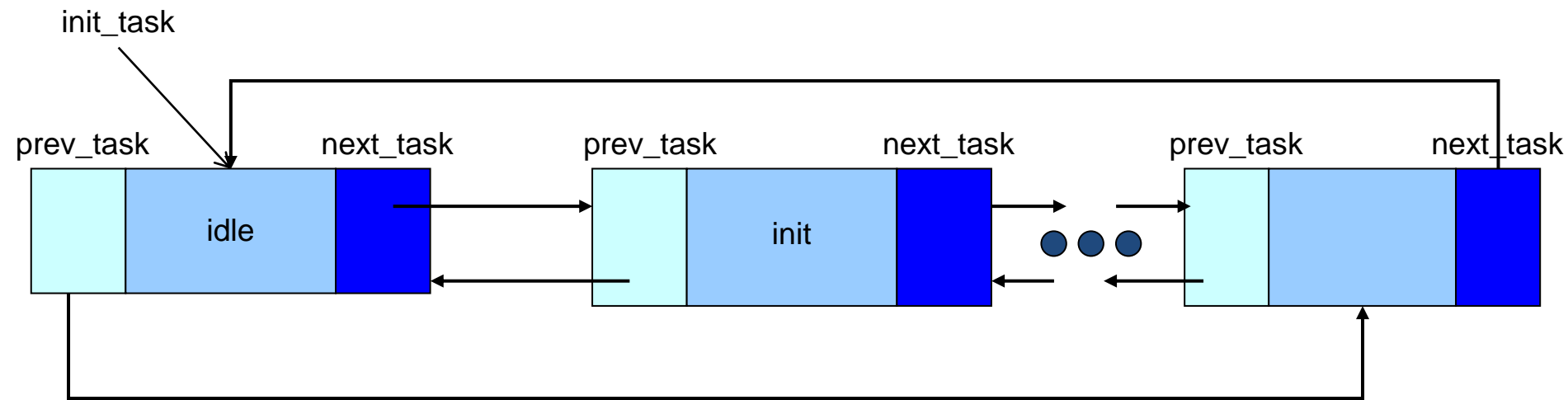
4. siblings: מצביע לרשימה
מקושרת של אחים, (תהליכים
הנוצרים ע"י אותו תהליך אב).



- כך תהליך יכול לאתר את בניו
לפי סדר יצירתם, למשל עבור
קריאת המערכת wait().

רשימת התהליכים

- מתארי כל התהליכים מחוברים ברשימה **מקושרת כפולה** **מעגלית** הקרויה **רשימת התהליכים** (Process List) באמצעות השדות `prev_task` ו-`next_task`.
- ראש הרשימה הוא המתאר של התהליך `idle` (מוצבע ע"י `init_task`).
- שאלה: למה הגיוני לעשות רשימה זו מקושרת **כפולה**?



פעולת על רשימת התהליכים

- איזו קריאת מערכת מוסיפה איבר (PCB) לרשימת התהליכים?

- `fork()`, כי היא יוצרת תהליך חדש.
- שימו לב: `execv()` אינה יוצרת תהליך חדש ולכן אינה מוסיפה איבר לרשימה.

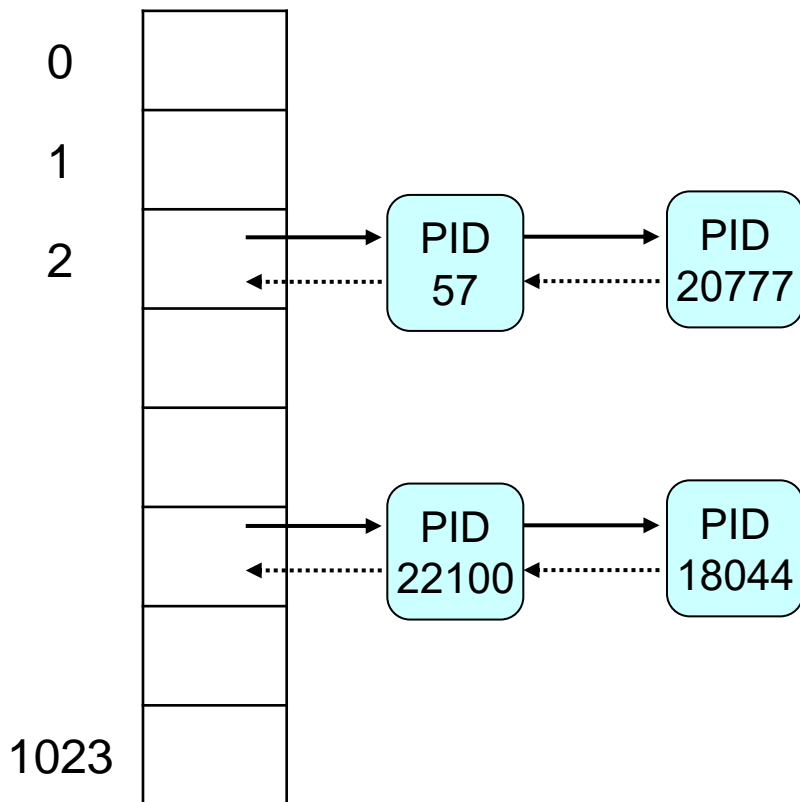
- איזו קריאת מערכת מוחקת איבר (PCB) מרשימת התהליכים?
- `wait()`, כי היא מוחקת תהליך שהסתיים.
- שימו לב: `exit()` אמנם מסיימת תהליך אבל אינה מוחקת אותו לגמרי (התהליך עובר למצב זומבי).

טבלת ערבול $PID \rightarrow PCB$

- חלק מקריאות המערכת, למשל `waitpid()`, מתייחסות לתהליכים ע"פ ה-pid שלהם.

- בעיה: חיפוש תהליך לפי pid ברשימה המקושרת של כל התהליכים הוא בסיבוכיות $O(n)$ כאשר n הוא מספר התהליכים במערכת.

- פתרון: הגרעין שומר טבלת ערבול (hash table) המאפשר לאתר תהליך לפי ה-pid שלו בסיבוכיות $O(1)$ בממוצע.



תורי תהליכים

תור ריצה לכל מעבד
תור המתנה לכל אירוע המתנה

מצב התהליך

- מצב התהליך נשמר בשדה **state** במתאר התהליך.
- משתנה בגודל 32 ביט המתפקד כמערך ביטים: בכל רגע נתון, בדיוק אחד מהביטים ב-state דלוק בהתאם למצב התהליך באותו זמן.

0000 0000 0000 0000 0000 0000 0000 0001

- המצבים האפשריים לתהליך בלינוקס הם:
 1. **TASK_RUNNING** – התהליך רץ או מוכן לריצה.
 - נאמר כי התהליך יזומן לריצה "בטווח הקצר".
 2. **TASK_ZOMBIE** – ריצת התהליך הסתיימה, אך תהליך האב של התהליך עדיין לא ביקש מידע על סיום התהליך באמצעות קריאה כדוגמת `wait()`.
- מתאר התהליך הוא הדבר היחיד שנותר ממנו.

מצב התהליך

3. **TASK_INTERRUPTIBLE** – המתנה "רדודה"

- התהליך ממתין לאירוע כלשהו אך ניתן להפסיק את המתנת התהליך ולהחזירו למצב TASK_RUNNING באמצעות שליחת **סיגנל** כלשהו לתהליך.
- זהו מצב ההמתנה הנפוץ. מתי נמצא תהליכים במצב זה?
- דוגמה 1: תהליך אב הממתין לסיום הבן (קריאת מערכת wait).
- דוגמה 2: דפדפן (web browser) מחכה לקבלת נתונים מהרשת (דף web) אבל אפשר לקטוע את המתנתו על-ידי סגירת חלון היישום, שגורמת לשליחת אות לסיום התהליך.

4. **TASK_UNINTERRUPTIBLE** – המתנה "עמוקה"

- התהליך ממתין לאירוע כלשהו אך לא ניתן להפסיק את המתנת התהליך באמצעות שליחת סיגנל לתהליך.
- מצב המתנה נדיר.
- דוגמאות: בשולי השקופית – דורשות חומר מתקדם בקורס.

5. **TASK_STOPPED** – ריצת התהליך נעצרה בצורה מבוקרת על-ידי תהליך אחר (בדרך-כלל debugger או tracer).

תור ריצה (runqueue)

- התהליכים המוכנים לריצה (מצב TASK_RUNNING) נשמרים במבנה נתונים הקרוי **runqueue** (תור ריצה).

- לכל ליבת מעבד יש תור ריצה (מבנה runqueue) משלה:

```
struct rq runqueues[NR_CPUS];
```

- בכל רגע נתון, תהליך יכול להימצא בתור ריצה אחד לכל היותר.

מדוע?

- מבנה הנתונים של תור ריצה מורכב מ:
 1. מערך של תורים לתהליכי זמן-אמת.
 2. עץ אדום-שחור לתהליכים רגילים.
- פרטים נוספים בתרגול על זימון התהליכים.

פעולות על תור ריצה

- הפונקציות `activate_task()`, `deactivate_task()` מכניסות ומוציאות תהליך מתור ריצה, בהתאמה.
- שימוש אפשרי לדוגמה: `wake_up_process()` הופכת תהליך ממתין (למשל במצב `TASK_INTERRUPTIBLE`) למוכן לריצה (מצב `TASK_RUNNING`):
 1. מוצאת את תור הריצה של המעבד הנוכחי.
 2. מוסיפה את התהליך לתור הריצה הזה באמצעות `activate_task()`.
 3. מסמנת צורך בהחלפת הקשר אם התהליך החדש בעדיפות גבוהה יותר מהתהליך שרץ כרגע על המעבד.

תור המתנה (wait queue)

- תהליך שממתין לאירוע כלשהו (מצבים TASK_INTERRUPTIBLE או TASK_UNINTERRUPTIBLE נמצא בתור המתנה (ואינו נמצא באף תור ריצה).
- לכל סוג אירוע יש תור המתנה נפרד, לדוגמה:
 - תור המתנה לכל סוג של פסיקת חומרה, למשל דיסק או שעון.
 - תור המתנה לכל משאב מערכת שיתפנה לשימוש. לדוגמה: ערוץ תקשורת שיתפנה כדי לשלוח דרכו נתונים.
 - תור המתנה לכל תהליך עבור סיום אחד הבנים שלו.
- תהליך יכול לעבור לתור המתנה רק באמצעות קריאת מערכת חוסמת (למשל read, wait, ... אשר מוותרת (yield) על המעבד.