אוניברסיטת חיפה
החוג למדעי המחשב
מערכות הפעלה – תרגול

תרגול מס' 2

Processes

1. מהו תהליך (Process)

תהליך הוא למעשה תוכנית אשר רצה במעבד.

או במילים אחרות: Process – a **Program** in **Execution**

1.1. ממה מורכב תהליך

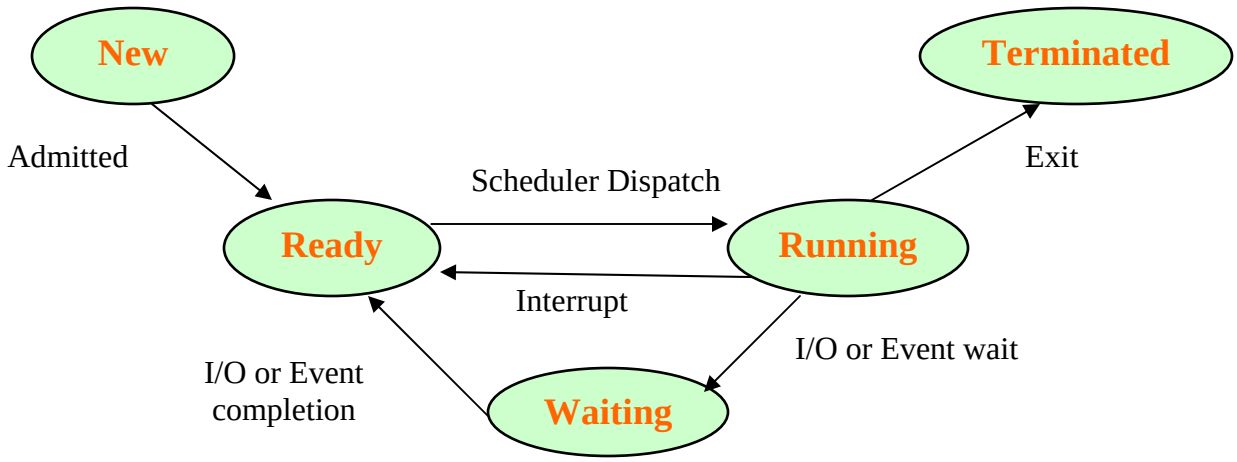
תהליך מורכב משלושה מרכיבים עיקריים:

- מונה התוכנית (Program Counter)
- מחסנית (Stack)
- אזור נתונים בזיכרון (Data Section)

1.2. מצבים של תהליך

תהליך יכול להימצא באחד מהמצבים הבאים:

- חדש (New) – התהליך כרגע נוצר
- במעבד (Running) – התהליך נמצא במעבד ומבוצעות פקודות מהתוכנית
- ממתין (Waiting) – התהליך נמצא בהמתנה (לרוב, ממתין למשאב)
- מוכן (Ready) – התהליך סיים להמתין למשאב וכרגע ממתין למעבד
- סיים (Terminated) – התהליך סיים את הריצה



1.3. Process Control Block

כל תהליך מכיל את האינפורמציה הבאה:

- Process State
- Program Counter
- CPU Registers
- …

1.4. Process Context

כל זמן שתוכנית כלשהי מתבצעת, הגרעין יוצר סביבה מתאימה לתוכנית זו. הסביבה כוללת את כל הנדרש כדי שהתוכנית תרוץ כאילו הייתה התוכנית היחידה במערכת. כאשר תוכנית חדשה נכנסת לביצוע נוצר תהליך. לכל תהליך יש Context ששומר את המצב הספציפי של ההרצה הנוכחית שלו וגם את כל הסביבה שנדרשת לתוכנית.

המרכיבים העיקריים של ה-Context:

- The text (program instructions) being run
- The memory used by the program being run
- The current working directory
- The files that are open and positions in the files
- Access control information
- Other various low-level information

כמו כן לכל תהליך יש מספר מאפיינים, כמו:

- מספר מזהה לתהליך (pid – process id)
- זהות המשתמש (uid – user id)
- Pid של תהליך האב
- המצב של התהליך (ריצה, מוכן לריצה, בהמתנה וכו')
- עדיפות לזמן התהליכים

ניתן לראות שחלק מהמרכיבים שצוינו לעיל מופיעים ב-PCB.

תהליך מתקשר עם הגרעין של מערכת ההפעלה בעזרת קריאות מערכת (System Calls). קריאות מערכת הינן אוסף מוגדר היטב של פונקציות שנתמכות ע"י הגרעין. קריאות מערכת מתבצעות ב-Kernel mode.

2. קריאות מערכת לניהול תהליכים

2.1. יצירת תהליך חדש ע"י *fork*

```
int fork( void );
```

יוצר תהליך חדש (ע״י שכפול). תהליך הבן (Child) הוא העתק של תהליך האב (Parent).

הערך המוחזר מ fork לתהליך האב הוא ה pid של הבן.

הערך המוחזר מ fork לתהליך הבן הוא 0.

PID=0 שמור לתהליך ה-Boot (שאח״כ משוכתב ל-Swapper)

אם נכשל, מוחזר (-1), ו errno מכיל את השגיאה.

2.2. מעבר לתוכנית אחרת ע"י *execl*

```
int execl( const char *path, char *arg0 [, char *arg1, ... , char *argn], (char *)0 );
```

הופך את התהליך הקורא לתהליך חדש. מתבצע Overlay על מרחב התהליך הקורא, עם תוכן הקובץ (הביצועי) שצוין, והביצוע ממשיך מנקודת הכניסה המצוינת בקובץ.

אין חזרה מפקודת exec מוצלחת - הקוד המקורי אבד.

path קובע מהיכן לטעון את התוכנית.

arg0 חייב להופיע, והוא מצביע למחרוזת הזהה ל-path (או המרכיב האחרון שלו).

argi מצביעים למחרוזות המרכיבות את רשימת הארגומנטים לתהליך החדש (יועברו כפרמטרי argc, argv לפונקציה main של התהליך החדש).

קיימות גם הקריאות:

execv – הארגומנטים מועברים במערך.

execp - ניתן לתת כפרמטר רק את שם התוכנית, ולא את ה- path המלא שלה. הקובץ של התוכנית ימצא ע״י חיפוש בספריות הנמצאות במשתנה PATH.

הקריאות execvp, execlp משלבות את האופציות שתוארו לעיל.

2.3. המתנה לתהליך בן ע"י *wait*

```
int wait( int *statptr );
```

מבצע Suspend לתהליך שקרא, עד שאחד מהבנים הישירים מסתיים (או עוצר כתוצאה מ-debugging).

אם הקריאה מוצלחת, מוחזר ה pid של הבן שסיים.

אם statptr אינו 0, 16 סיביות מידע המהוות את הסטאטוס של תהליך הבן נרשמות בכתובת ש - statptr מצביע עליה.

קיימת גם קריאת מערכת waitpid המקבלת בנוסף כפרמטר את ה- pid של תהליך בן מסוים.

התהליך הקורא יהיה במצב Suspend עד שהבן בעל ה-pid שצוין יסתיים.

2.4. המתנה ע"י *sleep*

```
unsigned sleep( unsigned  seconds );
```

משהה את התהליך מביצוע למשך מספר השניות המצוין בפרמטר. יתכן שההשהיה תהיה ארוכה יותר עקב פעילויות אחרות במערכת.

2.5. קבלת מס' pid של התהליך ע"י *getpid*

```
int getpid( void );
```

מחזיר את ה-pid של התהליך שקרא לפונקציה.

2.6. קבלת מס' pid של תהליך אב ע"י *getppid*

```
int getppid( void );
```

מחזיר את ה process-ID-של תהליך האב של התהליך שקרא.

2.7. דוגמאות

2.7.1. דוגמא מס' 1:

שימוש ב-exec ליצירת תהליך שונה מתהליך האב. המתנה לסיום תהליך הבן בעזרת wait, ובדיקת הסטאטוס המוחזר.

```
#include <stdio.h>
#include <unistd.h>
void main(void) {
  int status;
  if (0 == fork()) {
     execl("/bin/date", "date", NULL);
     exit(-1); /* error in the exec if we are here. */
  } else {
     wait(&status);
     printf("Parent is done. Status returned is %d\n", status);
   }
}
```

1. <u>דוגמא מס' 2:</u>

מציאת ה-pid של התהליך ושל תהליך האב

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

void main(void) {
  pid_t forkid;

  if(0 == (forkid = fork())) {
     printf("I am the child and my ID is %d\n", getpid());
     printf("I am the child and my parent's ID is %d\n", getppid());
   } else {
     printf("I am the parent and my ID is %d\n", getpid());
     printf("I am the parent and my child ID is %d\n", forkid);
   }
}
```

3. קריאות מערכת לטיפול בקבצים

3.1. ליצור או לפתוח קובץ חדש - *open*

```
#include <fcntl.h>
int open( const char *path, int flags [, int mode] );
```

פותח קובץ (path) על פי הרשום ב - flags שהוא צרוף (OR) של ערכים. למשל:

- אחד מ: O_RDONLY, O_WRONLY, O_RDWR
- O_NDELAY
- O_APPEND

הפרמטר השלישי אופציונלי. הוא מגדיר את זכויות הגישה לקובץ.

מחזיר File-Descriptor בו משתמשים בקריאות מערכת אחרות.

אם נכשל, מוחזר (-1), ו-errno מכיל את השגיאה.

<u>הערות</u>:

Errono משמש את הפונקציה perror(char *s) בהדפסת הודעת שגיאה מתאימה. errno אינו מתאפס לאחר קריאה מוצלחת!!!

3.2. יצירת קובץ חדש - *creat*

```
#include <fcntl.h>
int creat( const char *path, int mode );
```

יוצר קובץ חדש בשם הנתון, או מקצץ קובץ קיים. מחזיר File-Descriptor.

אם נכשל, מוחזר (-1), ו errno מכיל את השגיאה.

```
creat( path, mode );
```

שקול ל:

```
open( path, O_WRONLY | O_CREAT | O_TRUNC, mode );
```

3.3. ביצוע קריאה מהקובץ - *read*

```
int read( int fildes, void *buf, unsigned nbyte );
```

קורא nbyte בתים מהקובץ הקשור ל-fildes לתוך חוצץ ש-buf מצביע עליו.

מחזיר את מספר הבתים שנקראו בהצלחה (הערך 0 מוחזר כאשר הגיע סוף הקובץ).

אם נכשל, מוחזר (-1), ו errno מכיל את השגיאה.

3.4. ביצוע כתיבה לקובץ - *write*

```
int write( int fildes, const void *buf, unsigned nbyte );
```

כותב nbyte בתים מהחוצץ שמוצבע ע״י buf לתוך הקובץ הקשור ל - fildes.

מחזיר את מספר הבתים שנכתבו בהצלחה.

אם נכשל, מוחזר (-1), ו errno מכיל את השגיאה.

3.5. סגירה של file descriptor - *close*

```
int close( int fildes );
```

סוגר file-descriptor ומוחק אותו מהטבלה.

כשמצליח מחזיר 0, כשנכשל מחזיר (-1) והשגיאה נמצאת ב - errno.

4. תזמון תהליכים

4.1. מושגים

- **תזמון תהליכים** - קביעת התהליך הבא לריצה.
- **Time slice/Quantum** - פרק זמן מקסימלי על לתזמון מחדש.
- **זמן ריצה של תהליך** - משך הזמן בו התבצע התהליך עד כה.
- **Scheduler** - אובייקט במערכת ההפעלה הקובע ומריץ את התהליך הבא לריצה.
- **Clock tick** - יחידת זמן בסיסית במחשב.

- **עדיפות של תהליך** - לכל תהליך מיוחסת עדיפות אשר קובעת את מקומו בתור השייך לתהליכים המוכנים לריצה. העדיפות מורכבת מעדיפות קבועה ועדיפות דינמית המחושבת על פי זמן הריצה של התהליך (ככל שזמן הריצה גדול יותר יורדת העדיפות הדינמית).

בשקפים של ההרצאה ניתן למצוא תרשים המציג סכימה כללית של תזמון תהליכים.

4.2. Schedulers

ישנם 3 סוגים של מתזמנים במערכת:

4.2.1. Long Term Scheduler

תפקידו העיקרי לטעון תהליכים מהדיסק לזיכרון. בנוסף הוא לא עובד כל הזמן (ה- Intermediate Scheduler עובד יותר). ה-Long Term אחראי על דרגת המקביליות במעבד – הוא מחליט כמה תהליכים יהיו בזיכרון, מתי להזיז תהליכים לדיסק וכו'.

4.2.2. Short Term Scheduler

תפקידו להעביר תהליכים המצויים בזיכרון בין המצבים השונים.

4.2.3. Intermediate Term Scheduler

תפקידו להחליף תהליכים מהזיכרון לדיסק (מאוד דומה ל-Long Term Scheduler).

4.3. קביעת עדיפותו של תהליך

קיימות שלוש רמות עדיפות:

1. עדיפות Kernel גבוהה - לתהליכים הרצים ב Kernel mode.
2. עדיפות Kernel נמוכה - לתהליכים הרצים ב Kernel mode.
3. עדיפות User - לתהליכים הרצים ב User mode.

בכל Clock tick מתעדכן מונה בשם CPU השייך לתהליך הנוכחי אשר מתבצע במחשב

דוגמא לנוסחת חישוב עדיפותו של תהליך ב User mode:

Priority = CPU/4 + Base_priority

- **ככל שהתוצאה גבוהה יותר כך העדיפות האמיתית נמוכה יותר.**
- בתהליכי User ה-Base_priority גדול מערך עדיפות ה-Kernel הגדולה ביותר ולכן תמיד ירוצו תהליכי Kernel **לפני** תהליכי User.

- ניתן (אם רוצים) להריץ תהליכים בעדיפות נמוכה. ניתן לקרוא לפונקצית המערכת nice(val) ולשנות את ה Base_priority.

4.4. דוגמה – זמני ריצה של תהליכים

במהלך הדוגמה נתון כי :

Time Slice = 60 Clock Ticks Base_priority = 60

שני תהליכים A ו B רצים שניהם ב User mode.

שני התהליכים התחילו באותו זמן כאשר תהליך A נכנס ראשון לריצה.

הניחו ש- Context switch הינו מיידי.

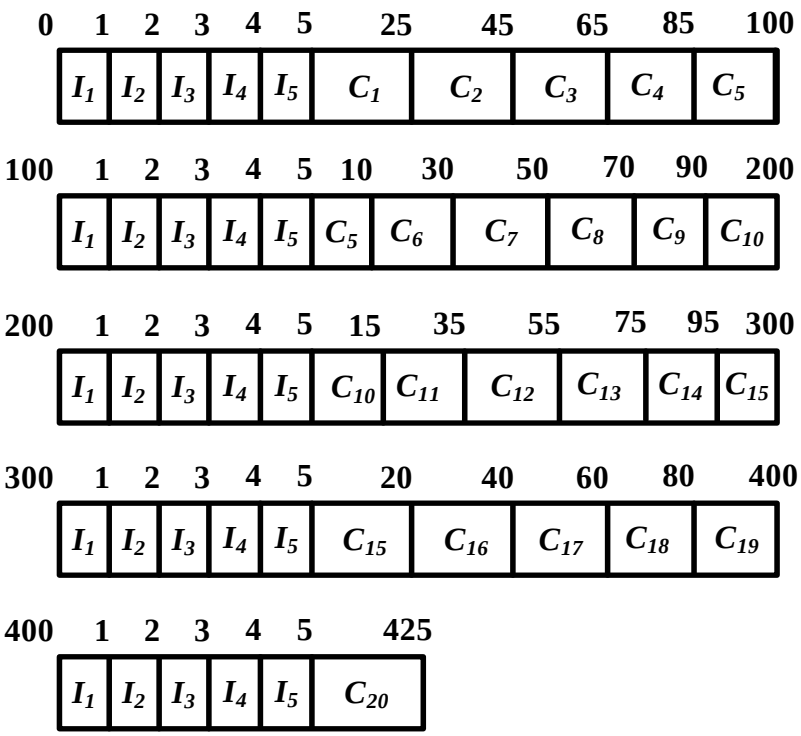
Clock Ticks	Prior A	CPU	Prior B	CPU
0	60	0	60	0
10	<-	10		
20	<-	20		
30	<-	30		
40	<-	40		
50	<-	50		
60	75	60	60	0
70			<-	10
80			<-	20
90			<-	30
100			<-	40
110			<-	50
120	60	0	75	60
130	<-	10		
140	<-	20		
150	<-	30		
160	<-	40		
170	<-	50		
180	75	60	60	0
190			<-	10
200			<-	20

<u>הערה</u>: הדוגמא הנ"ל היא דוגמא מאור מופשטת ולא אופיינית. לצורך חישוב אמיתי של עדיפות נלקחים בחשבון מספר גורמים כגון Base Priority, זמן CPU, זמן שחלף מאז קיבל התהליך זמן CPU לאחרונה וכו'.

4.5. דוגמה - תזמון תהליכים:

במערכת קיימים 5 תהליכים מסוג I, ו- 20 תהליכים מסוג C, כאשר:

- כל תהליך מסוג I מבצע 4 איטרציות, כאשר כל אחת מהן כוללת חישוב הצורך יחידת זמן 1, ולאחריו, פעולת I/O למשך 99 יחידות זמן. לאחר מכן מבצע התהליך חישוב אחרון הצורך יחידת זמן 1.
- כל תהליך מסוג C מבצע חישוב הצורך 20 יחידות זמן.
- כל התהליכים מוכנים לריצה בזמן 0, כאשר תהליכי I הגיעו לתור המוכנים <u>לפני</u> תהליכי C.
- המערכת מסוגלת לטפל בפעולות ה- I/O של תהליכי I במקביל.
- זמן ה- Context Switch זניח.

<u>שאלה</u>: מצאו סדר זימון אופטימלי מבחינת זמן סיום פעולת התהליכים. ציינו את זמן סיום פעולת התהליך האחרון מכל סוג.

<u>פתרון</u>: