מערכות הפעלה - הרצאה 5

שרון מלטר, אתגר 17

8 ביולי 2024

1 Chapter 3

תזכורת: *call system* זוהי כל בקשה לפעולה של הקרנל על ידי שכבה עליונה יותר.

outline:

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

מטרות הפרק:

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

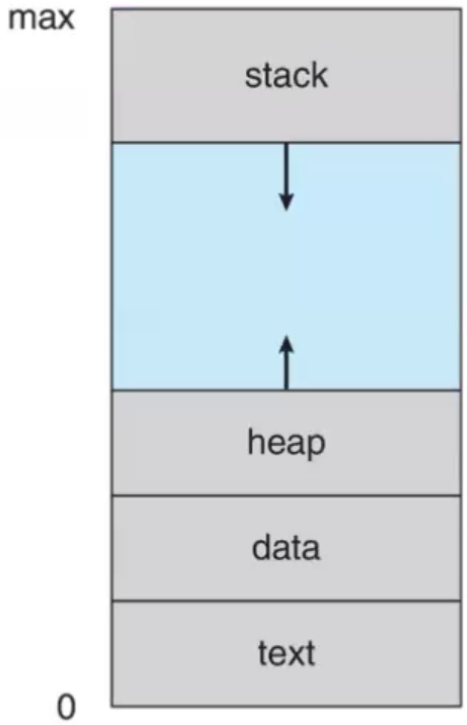
1.1 Processes

מהו הקונספט של תהליך?
כאשר אנו כותבים תכנית, היא רק יושבת על ה- *disk hard*. אך כאשר אנחנו מריצים אותה, היא הופכת לתהליך.

כמו כל קוד את אותה התכנית אפשר להריץ כמה פעמים, אך בכל פעם התהליך יהיה תהליך נפרד. בקיצור, תהליך הוא תכנית בהרצה וכמובן שלתוכנה יכולים להיות מספר תהליכים. ניתן להתחיל את הרצת התכנית למשל בעזרת הקלקת עכבר על אלמנטים של GUI תהליך חייב לרוץ באופן סדרתי, לא ניתן לבצע הרצה מקבילית של שתי פקודות של אותו תהליך.
התהליך מורכב מ־5 חלקים:

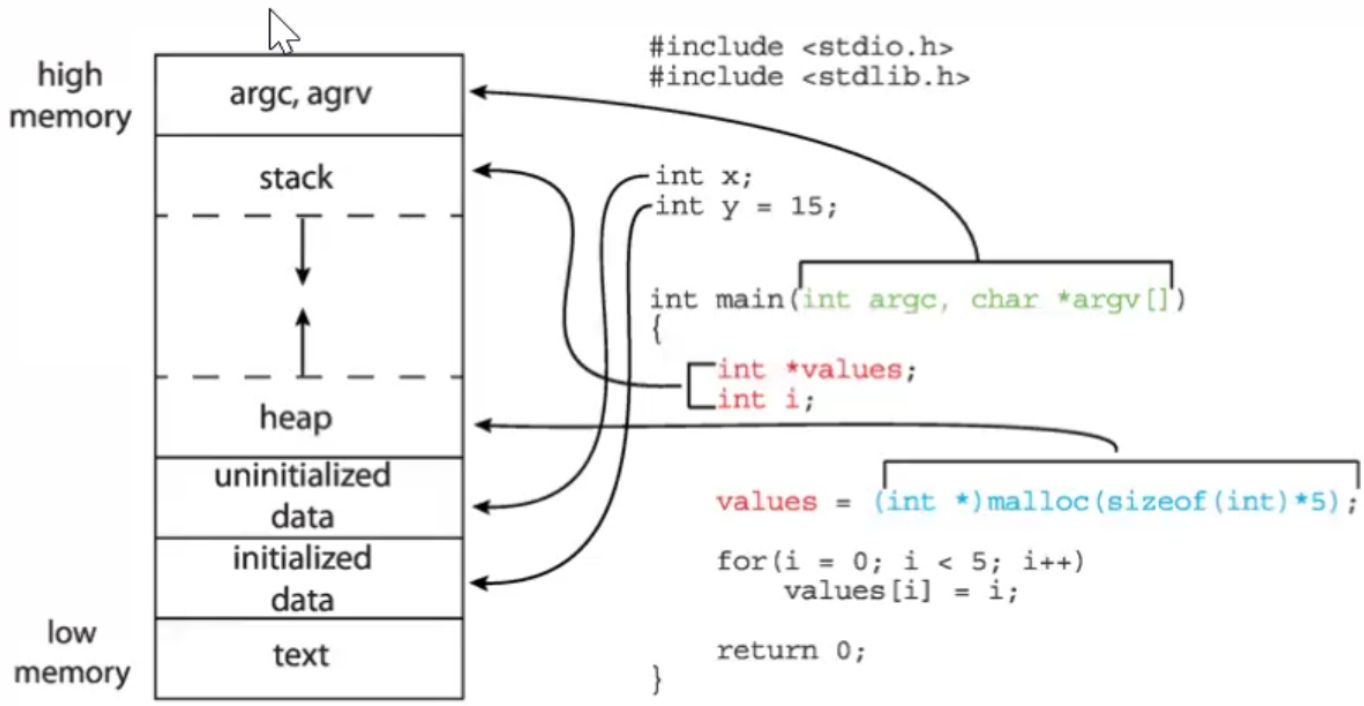
1. קוד התכנית, נקרא גם $text\ section$
2. האקטיביטי הנוכחית, שכוללת $program\ counter$ ואת הרגיסטרים של התהליך.
3. מחסנית שבה נתונים זמניים (למשל פרמטרים של פונקציות, משתנים לוקאליים, כתובתו מוחזרות (שאינן הכתובות האמיתיות בזיכרון, הן רק כתובת וירטואלית))
4. $Data\ section$ שמכיל משתנים גלובאליים.
5. $Heap$ שבה זיכרון שהוקצה דינמית בזמן הריצה.

זהו איור של תהליך, כפי שהוא נמצא בזיכרון; כצפוי, נשמר מקום פנוי עבור $Heap$, מכיוון שלא ניתן לדעת



מהו גודל הזיכרון הנדרש לפני ההרצה. עם זאת, גם גודל הזיכרון הנדרש ל־ $stack$ יכול להשתנות־ למשל כאשר קוראים קריאה רקורסיבית.

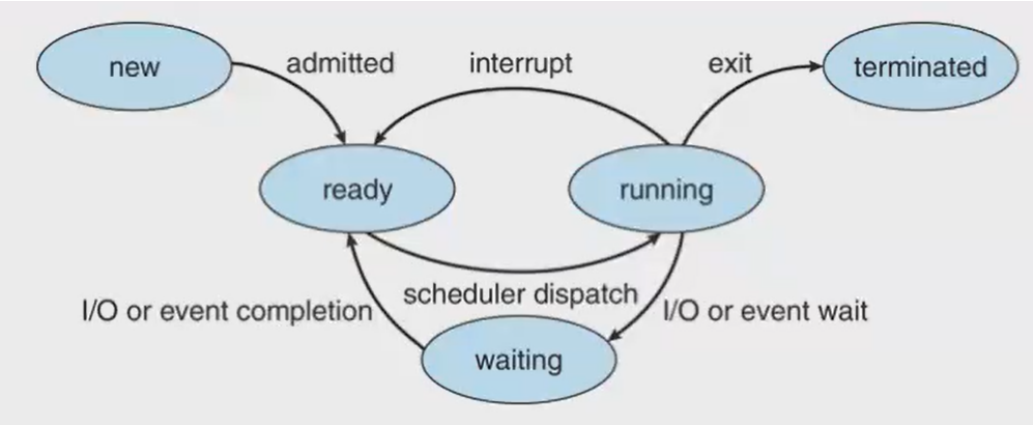
נראה תרשים המייצג את הזיכרון של תכנית ב־ C; מעבר לנתונים שהזיכרון שומר, יש לשמור עבורו גם את



מצב התהליך ־ *Process State*. כאשר תכנית רצה, היא משנה את מצבה. להלן רשימת המצבים האפשריים של תהליך:

1. *New* ־ כאשר התהליך נוצר.
2. *Running* ־ כאשר פקודות התהליך רצות.
3. *Waiting* ־ התהליך מחכה לכך שאירוע כלשהו יקרה.
4. *Ready* ־ התהליך מחכה לרוץ במעבד.
5. *Terminated* ־ התהליך סיים לרוץ (לא בהכרח מתקלות, אלא גם אחרי ביצוע תקין של כל הפקודות)

להלן דיאגרמה של המעברים בין מצבי תהליך; כעת נראה כיצד כל רכיבי הזיכרון נשלטים ־ באמצעות ה־



Process Control Block (PCB). זהו בלוק המכיל את כל המידע על תהליך מסוים. יכול להיקרא גם *Block Task Control*. לפי הגדרה, בלוק זה מכיל את הנתונים הבאים:

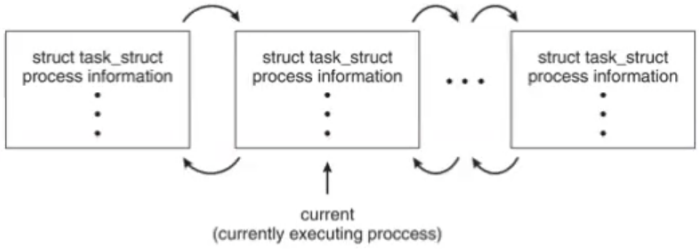
- *Process state*

- $Program\ counter$
- $CPU\ registers$ - כל הרגיסטרים המסוימים של התהליך.
- $CPU\ scheduling\ information$ - עדיפויות, מצביעים של תור לוח הזמנים (מצביע בתור בו מחכים התהליכים להגיע ל- CPU)
- $Memory-management\ information$ - הזיכרון שהוקצה בשביל התהליך.
- $Accounting\ information$ - ה- CPU-ים שמשומשים, הזמן ב- $clock$ מאז תחילת הרצת התהליך, גבולות זמן מותר.
- $I/O\ status\ information$ - התקני I/O שמוקצים לתהליך, שרשרת של קבצים פתוחים.

כעת נעבור לדוגמה.
להלן ייצוג של תהליך ב- $Linux$;

Represented by the C structure `task_struct`

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this
process */
```



כפי שניתן לראות, כל נתוני התהליך שמורים במבנה זה.
כעת נלמד מה הם חוטים, $threads$, שמשמשים לפירוק של תהליך.

1.2 חוטים

עד כה, התייחסנו לתהליך כאל בעל חוט יחיד שרץ. אך נחשוב מה קורה אם ישנם מספר $program\ counters$ לכל תהליך: נוכל להשתמש במספר חוטים ולבצע פקודות שונות במקביל. עם זאת, נצטרך גם אחסון לפרטי כל החוטים. עוד על נושא זה בפרק 4. עכשיו נעמיק בכיצד בוחרים איזה תהליך מקבל CPU.

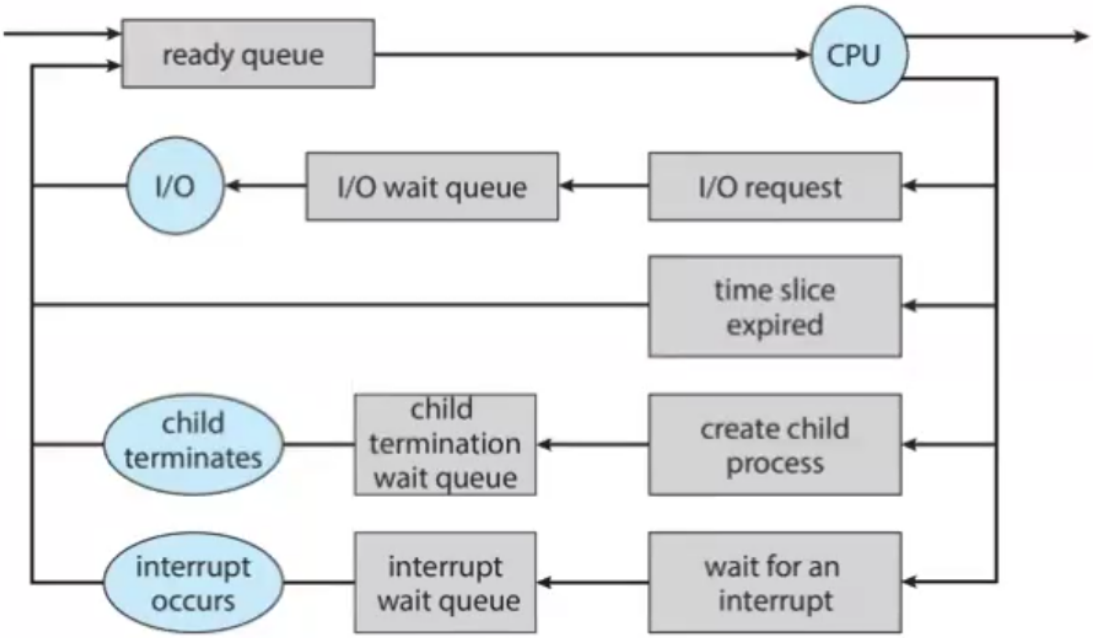
1.3 תזמון תהליכים

ה- $Process\ scheduler$ בוחר מבין התהליכים המוכנים מי יהיה הבא שירוץ ב- CPU. המטרה היא למקסם את השימוש ב- CPU כך שיהיה עסוק כמה שיותר ולכן גם למזער את הזמן שלוקח להחליף בין התהליכים שמשתמשים ב- CPU. על מנת לנהל את התהליכים שיכולים להשתמש ב- CPU, ישנם שתי תורים בהם הם שמורים.

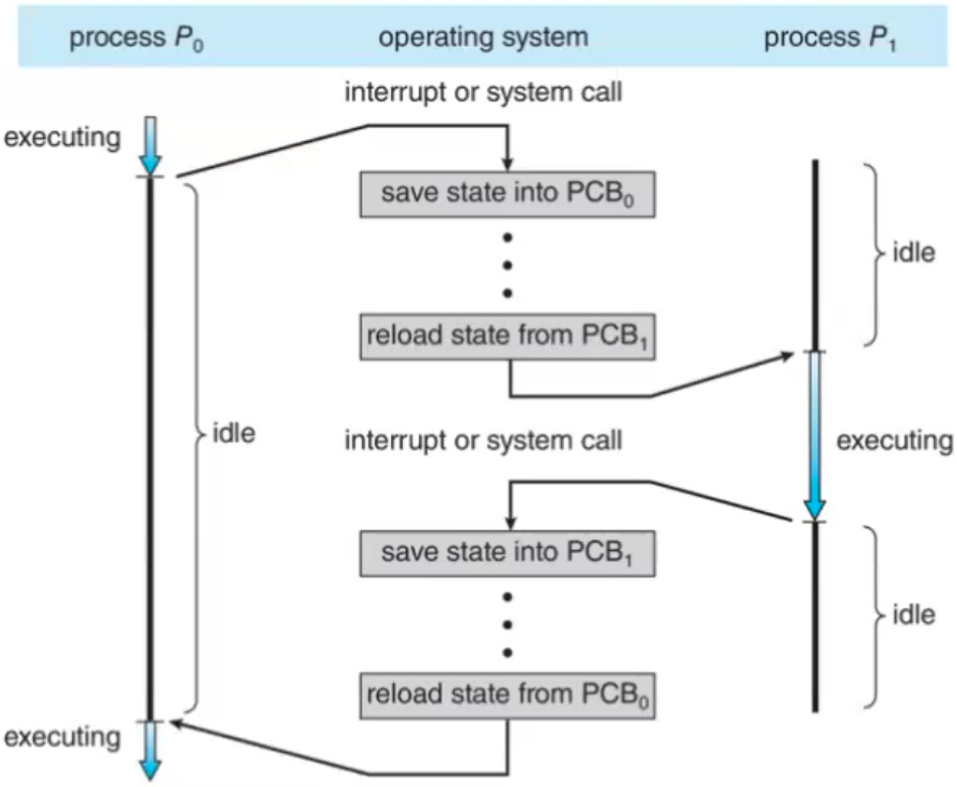
- $Ready\ queue$ - סט של כל התהליכים שנמצאים בזיכרון הראשי ומחכים לרוץ.

- $Wait\ queues$ - סט של תהליכים אשר מחכים לאירוע (למשל I/O)

תהליכים כמובן יכול לנוע בין התורים השונים. בהמשך ישנו תרשים של תזמון התהליכים; וכך מחליפים בין שני



תהליכים ב- CPU;



כאשר התהליך הראשון עוצר את ריצתו, הוא נשמר ל- PCB שלו, כך שניתן יהיה להמשיך אותו מאותו החלק כאשר הוא יחזור (אם איננו סיים) ואז מעלים את ה- PCB של התהליך השני.

עם זאת, ישנו מושג חדש שיש להכיר־ $Context\ Switch$. כאשר ה־ CPU מחליף תהליך, המערכת חייבת לשמור את ה־ $state$ של התהליך הקודם ולהעלות את המצב של התהליך החדש, בעזרת $Context\ Switch$. (קונטקסט של תהליך מיוצג ב־ PCB כמובן) הזמן של החלפת קונטקסט הוא $overhead$ טהור, לא ניתן לעבוד במערכת באותו זמן. לצערנו, ככל שמערכת ההפעלה מסובכת יותר, כך זמן ה־ $Context\ Switch$ גדל. עם זאת, אורך הזמן תלוי גם בתמיכת החומרה. למשל, כאשר חומרה מספקת מספר רגיסטרים לכל תהליך, ישנם מספר קונטקסטים שצריכים להיות מועלים בנפרד.
נעבור למה עוד ניתן לעשות עם תהליכים.

1.4 פעולות על תהליכים

ישנן שתי פעולות אשר מערכת ההפעלה חייבת לספק לתהליכים.

1.4.1 יצירת תהליך

תהליך הורה יוצר תהליכים ילדים, שבתורם יוצרים תהליכים אחרים, כך שנוצר עץ של תהליך. באופן כללי, כל תהליך ניתן לזיהוי על ידי $process\ identifier\ (pid)$. שימי לב שמספר זהות זה איננו יכול להיות שלילי, כך שאם הוא שלילי ניתן להסיק כי התרחשה תקלה. כמו כן, כאשר $pid == 0$, ניתן לדעת שאנו כעת מריצים את תהליך הילד ואם $pid > 0$ אז זהו מספר הזהות של תהליך ההורה (זהו פשוט מספר הזהות של ההורה) כל הורה יכול לקרוא לפקודה $wait()$ על מנת לחכות שהילד יסיים את ריצתו (ניתן לזהות מתי הילד מסיים אותה בעזרת ה־ pid שלו) ישנן מספר דרכים לשתף משאבים בין הורים לילדים;

1. הורה וילד משתפים את כל המשאבים.
2. הורים חולקים חלק מהמשאבים שלו עם הילדים שלו.
3. הורה וילד לא חולקים אף משאב.

כמו כן יש שתי דרכים להרצת הורה וילדו;

1. ההורה והילד רצים באופן מקביל.
2. ההורה מחכה עד שהילד מסיים את ריצתו.

ישנם שני סוגים של ילדים להורים;

1. הילד הוא העתק של ההורה. כלומר, הם חולקים את אותה התכנית. עם זאת, ה־ $program\ counter$ של הילד יהיה זהה לזה של ההורה שלו לאחר שקרא ל־ $fork()$
2. תכנית אחת מועלית אל הילד.

ב־ $Unix$, כאשר קוראים ל־ $fork()$ נוצר ילד עם תכנית שזהה לתכנית האב. אם קוראים ל־ $fork()$ ולאחר מכן ל־ $exec()$, אזי הילד מקבל תכנית אחרת.

1.4.2 סיום תהליך

לאחר שתהליך מריץ את הפקודה האחרונה שלו, הוא מבקש ממערכת ההפעלה למחוק אותו באמצעות קריאת המערכת $exit()$. קריאה זו מחזירה את נתון המצב מאותו התהליך להורה שלו (דרך $wait()$) והמשאבים של התלך משוחריים על ידי המערכת.
ייתכן שהורה יסיים את הרצת הילדים שלו באמצעות קריאת המערכת $abort()$. חלק מהסיבות לעשות זאת הן;

- הילד זקוק ליותר מכל המשאבים הניתנים.
- המשימה שניתנה לילד איננה נחוצה יותר.
- ההורה מבצע $exit()$ והמערכת איננה מרשה לילד להמשיך לרוץ אם ההורה שלו סיים. מקרה זה נקרא סיום מדורג $(casacading\ termination)$, בו כל הילדים והנכדים ושאר הצאצאים של תהליך ההורה מופסקים.

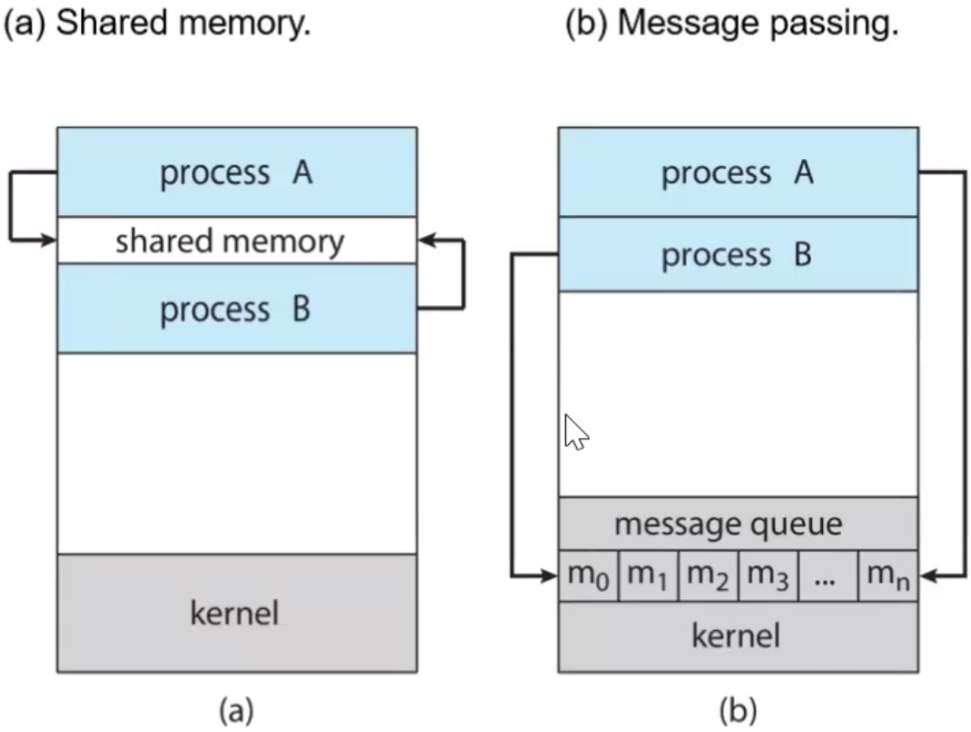
המצב ההפוך לסיטואציה השנייה שתוארה הוא המצב בו ההורה קורא ל־ $wait()$ ומחכה עד שהילד שלו סיים לרוץ. הקריאה לפונקציה זו מחזירה את הסטאטוס וה־ pid של אותו תהליך ילד; $pid = wait(\&status)$. אם אף הורה לא חיכה לתהליך כלשהו עד שהוא הסתיים (לכל תהליך מלבד לראשון יש הורה), כלומר אף תהליך לא קרא עבורו ל־ $wait()$, התהליך הופך לזומבי. אם ההורה שלו הסתיים בלי לקרוא ל־ $wait$, תהליך הילד הופך ליתום.
עד כאן תקשורת הרלוונטית רק בין הורים וילדים. נעבור לדבר על תקשרות כללית בין תהליכים.

1.5 תקשורת בין תהליכים

תהליכים שונים במערכת יכולים להיות בלתי־תלויים או משתפי־פעולה. תהליכים שמשתפים פעולה יכולים כמובן להיות מושפעים או להשפיע על תהליכים אחרים, למשל על הדאטה שבה הם משתמשים.
להלן מספר סיבות לשיתוף פעולה בין תהליכים;

- חלוקת מידע
- האצת חישוב
- מודולאריות
- נוחות

תהליכים שמשתפים פעולה זקוקים לתקשורת בין־תהליכית, או $interprocess\ communication\ (IPC)$. ישנם שני מודלים של IPC - זיכרון משותף והעברת מסרים. נייצג אותם בתרשימים הבאים; תחילה, נגדיר בעיה אשר



תסייע לנו להבין תקשורת בין תהליכים.

1.5.1 בעיית הצרכן־צורך $(Producer - Consumer\ Problem)$

זהו משל עבור תהליכים המשתפים פעולה. התהליך היצרן מספק מידע שמשמש את התהליך הצרכן. ישנן שתי גירסאות למערכת יחסית זאת;

- $unbounded - buffer$ - במקרה זה אין גבול לגודל של הבאפר, בו היצרן מציב מידע והצרכן קורא אותו. לכן היצרן לעולם לא יצטרך לחכות לכך שיהיה מקום פנוי בבאפר, אך הצרכן מחכה אם אין מה לצרוך בבאפר.
- $bounded - buffer$ - במקרה זה גודל הבאפר מוגבל. לכן היצרן צריך לחכות אם הבאפר מלא במידע והצרכן צריך לחכות אם אין מה לקרוא בבאפר.

באמצעות משל זה, נלמד איך ליישם פתרונות לשני מערכות היחסים שראינו.

1.5.2 זיכרון משותף

במקרה זה ישנו מרחב בזיכרון שהינו משותף למספר תהליכים. שימי לב־ התקשורת היא בשליטת התהליכים עצמם, ולא על ידי המערכת. אחד האתגרים הוא לספק מכניקה שיאפשר לתהליכי היוזר לתזמן את הפעולות שלהם על הזיכרון המשותף, כך שאחד לא יקרא ממנו בזמן שהשני משנה אותו.
נבין כיצד מממשים את הבאפר שבו שמור זיכרון משותף. בדוגמה הבאה למשל, בבאפשר ישנם 9 אלמנטים שניתן לשמור בבאפר; איך מבוצעת פרוצדורת היצרן?

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

בודקים אם המקום הבא אליו ניתן להכניס אלמנט הוא המקום הבא שממנו הצרכן אמור לקרוא. אם כן, אז האלמנט שאמור להיקרא יימחק. לכן לא ניתן לכתוב במקרה זה, כך שאין להכניס אלמנט כעת כך שתתאפשר כתיבה למיקום הבא. אך כן ניתן בשאר המקרים.

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

איך מבוצעת פרוצדורת הצרכן?
בודקים האם הקריאה כעת מבוצעת מאלמנט שעוד אמור להיכנס (האם $out == in$). אם כן, כפי שהוסבר, תיתכן תקלה. לכן ניתן לקרוא רק בכל מקרה אחר.

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

אחת הבעיות המרכזיות בשיתוף זיכרון, הוא *Race Condition*. זהו מצב בו שני תהליכים מתרחשים באופן מקביל, ולא ניתן לדעת מי מהם יתבצע קודם או שחלק מהראשון מתבצע ואז חלק מהשני... להלן דוגמה; אפשר

counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

לראות ש־ *race condition* עלול לגרום להתנהגות בלתי צפויה.

1.5.3 העברת מסרים

בסוג זה של תקשורת אין פרמטרים משותפים. במקום זאת, ישנן שתי פעולות שה־ IPC מספק ־ $send(message)$, $receive(message)$ הגודל של המסר $(message)$ הוא קבוע או תלוי בפרמטר. אם תהליכים P, Q רוצה לתקשר, הם צריכים ליצור לינק תקשורת $(communication\ link)$ ביניהם ולהחליף מסרים באמצעות $send$ ו־ $receive$.
להלן מספר אתגרי אימפלמנטציה:

- כמה לינקים נוצרים?
- האם לינק יכול להיות משויך ליותר משני תהליכים?
- כמה לינקים יכולים בין כל שני תהליכים?
- מהי הקיבולת של הלינק?
- האם הגודל של המסר שהלינק יכול להעביר הוא קבוע או ניתן להבעה כפרמטר?
- האם לינק צריך להיות חד או דו כיווני?

לכל לינק תקשורת יש תכונות פיזיות ולוגיות אפשריות.
תכונות פיזיות:

- זיכרון משותף
- $Hardware\ bus$
- $Network$

תכונות לוגיות:

- $Direct$ או $Redirect$
- סינכרוני או אסינכרוני
- $Automatic$ או $Explicit\ buffering$

נסביר תכונות אלה.
$Direct\ Communication$ היא תקשורת בה תהליכים חייבים לקרוא אחד לשני באופן ישיר $(explicit)$. למשל, אם רוצים לשלוח מסר מתהליך Q לתהליך P קוראים לפונקציה $send(P, message)$. כדי לקבל מסר מהתהליך Q יש לקרוא לפונקציה $receive(Q, message)$. תכונות של לינק תקשורת ישירה:

- לינקים נוצרים באופן אוטומטי.
- כל לינק מזוהה עם זוג יחיד של תהליכים.
- בין כל שני תהליכים קיים בדיוק לינק יחיד.
- הלינק יכול להיות חד־כיווני, אך לרוב הוא דו־כיווני.

נעבור ל־ $Indirect\ Communication$.
המסרים מכוונים ומקבלים אותם מתיבות דואר $(mailboxes)$, המכונות גם נמלים $(ports)$. לכל נמל יש id מיוחד ותהליכים יכולים לתקשר רק אם יש להם נמל משותף. תכונות של לינק תקשורת עקיפה:

- לינק נוצר רק אם לתהליכים יש נמל משותף.
- לינק יכול להיות מזוהה עם תהליכים רבים.
- כל זוג של תהליכים יכול לחלוק מספר לינק.
- לינק יכול להיות חד־כיווני או דו־כיווני.

פעולות בתקשורת עקיפה:

- יצירת נמל חדש.

- שליחת וקבלת מסר דרך נמל. דרך הפונקציות $send(A, message)$, $receive(A, message)$ כאשר A היא נמל.
- מחיקת נמל.

עם זאת, בטח שמת לב למסקנה מעניינת; תהליך יכול לשלוח מסר לנמל, ולא לדעת איזה תהליך יקבל אותו. ישנן מספר דרכים להתייחס להחלטה מי מקבל את המסר;

- לתת ללינק להיות מזוהה רק עם לכל היותר זוג תהליכים.
- לתת לתהליך אחד בכל זמן לקרוא לפעולה $receive$
- לתת למערכת להחליט באופן שרירותי מי מקבל את המסר. התהליך המוסר מיודע מי הוא התהליך שקיבל את המסר ששלח.

מעבר לכך, שיטת העברת המסרים יכולה להיות סינכרונית או אסינכרונית.
כאשר שיטת העברת מסרים היא $Blocking$, היא נחשבת סינכרונית.

- $Blocking\ send$ - כאשר השולח חסום עד שהמסר הקודם ששלח נקרא.
- $Blocking\ receive$ - כאשר התהליך שמקבל מסר חסום עד שמתקבל מסר זמין.

לעומת זאת, שיטת $Non-blocking$ נחשבת אסינכרונית.

- $Non-blocking\ send$ - השולח מוסר את המסר וממשיך.
- $Non-blocking\ receive$ - מקבל המסר מקבל מסר חוקי, או מסר ריק $Null$

כמו כן ישנן מספר קומבינציות אפשריות. אם גם השולח וגם המקבל הם $blocking$, יש לנו $rendezvous$ ('פגישה' בצרפתית)
נעבור למימוש של פרוצדורות היצרן והצרכן. כעת נעבור לדבר על הידיד החשוב שעזר לנו עד כה, הבאפר.

- Producer

```
message next_produced;
while (true) {
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed)

  /* consume the item in next_consumed */
}
```

1.5.4 $Buffering$

הבאפר הינו תור המסרים המחובר ללינק. ניתן ליישמו באחת משלוש דרכים;

1. $Zero\ capacity$ - אף מסר לא מחובר ללינק. השולח חייב לחכות למקבל $(rendezvous)$
2. $Bounded\ capacity$ - מספר סופי n של מסרים, השולח חייב לחכות אם התור מלא.
3. $Unbounded\ capacity$ - אורך התור הוא אינסופי. השולח אף פעם לא צריך לחכות.