פרק 3
תהליכים

מרצה: אליאב מנשה

מבוא

בעבר:

מערכות המחשב (ומערכות ההפעלה) הראשונות אפשרו להריץ תכנית אחת (בלבד) בו זמנית.

לתכנית המורצת הייתה שליטה מלאה על המערכת והמשאבים.

כיום:

מ"ה מאפשרת העלאה לזיכרון והרצה של מספר תכניות והרצה במקביל.

מחייב שליטה הדוקה יותר של מערכת ההפעלה.

מטרות השיעור

להציג את ה-"תהליך" (process)

- תכנית בהרצה (program in execution)

לתאר מספר מאפיינים (features) של תהליכים, ובכלל זה:

- תזמון
- יצירה
- עצירה
- תקשורת

ה- Process

בקורס נשתמש במושגים job ו- process כחליפיים

- Jobs - בד"כ מתייחס ל – batch systems
- Process – בד"כ למערכות שהם time-shared

התהליכים הרצים במערכת:

- תהליכים של מערכת ההפעלה המריצים system code
- תהליכים של משתמשי הקצה המריצים user code

Process – "תכנית בהרצה"

- Program (executable) – ישות פאסיבית השוכנת על הדיסק
- Process – ישות אקטיבית אשר משנה באופן רצוף את מצבה

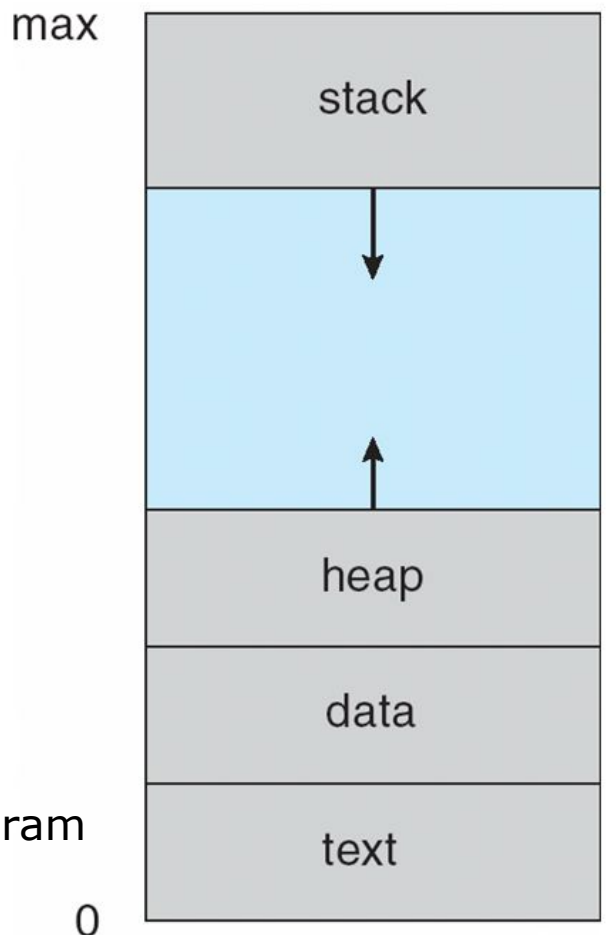
הרצת תהליך מתבצעת באופן סדרתי (sequential)

התהליך

התהליך כולל:

- Program counter
- Stack and Heap
- Text and data section

Process



Usually temporary data (such as function parameters, return addresses, and local variables) – stored for a short amount of time in memory

Dynamically allocated during process run-time

Global variables

compiled code of the program

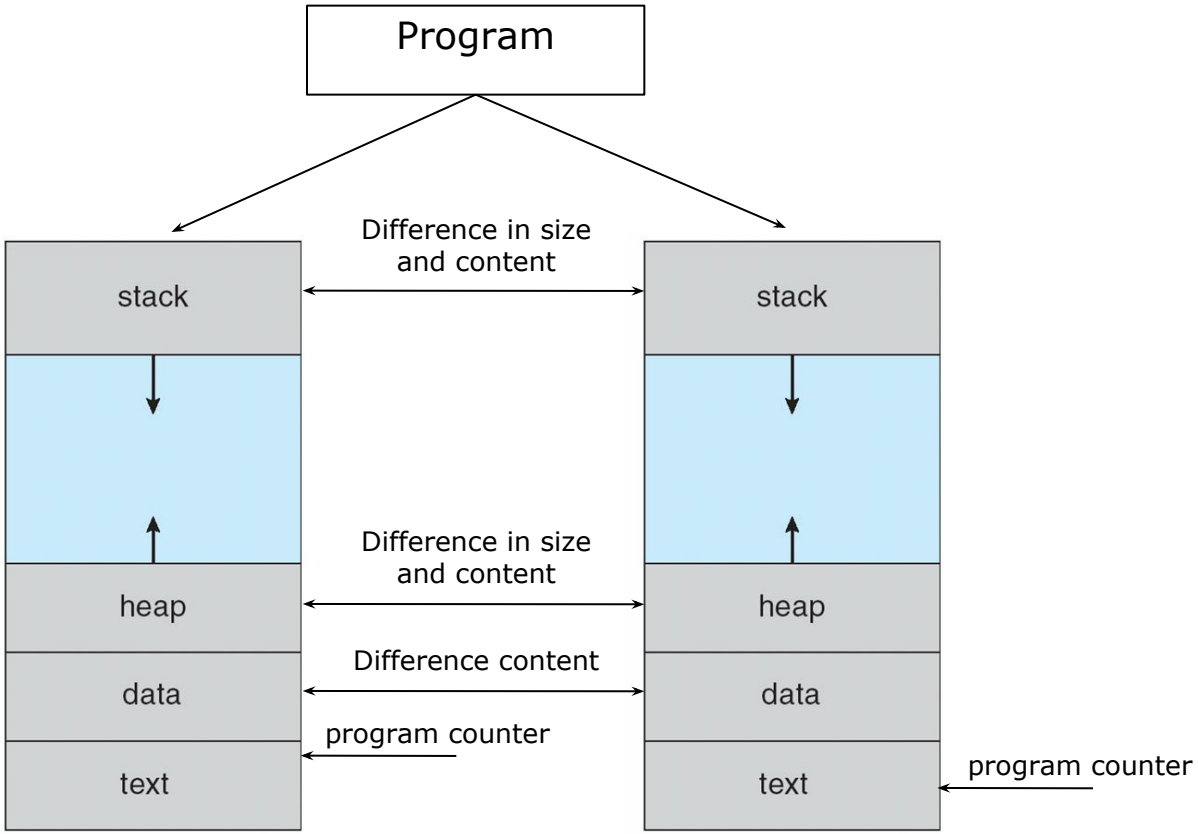
```
void foo()
{
  int x; <<< x is on the stack
  char *ptr = new char[255]; <<< 255 characters are allocated in the heap
                              but ptr object is on the stack

int array[255]; <<< 255 ints are all on the stack
}
```

Program and Process

על בסיס אותה תכנית ניתן להריץ שני תהליכים או יותר (במקביל) ,

כשלכל אחד יש execution sequence משלו



Process State

במהלך ריצתו משנה תהליך את מצבו (state):

momentary states

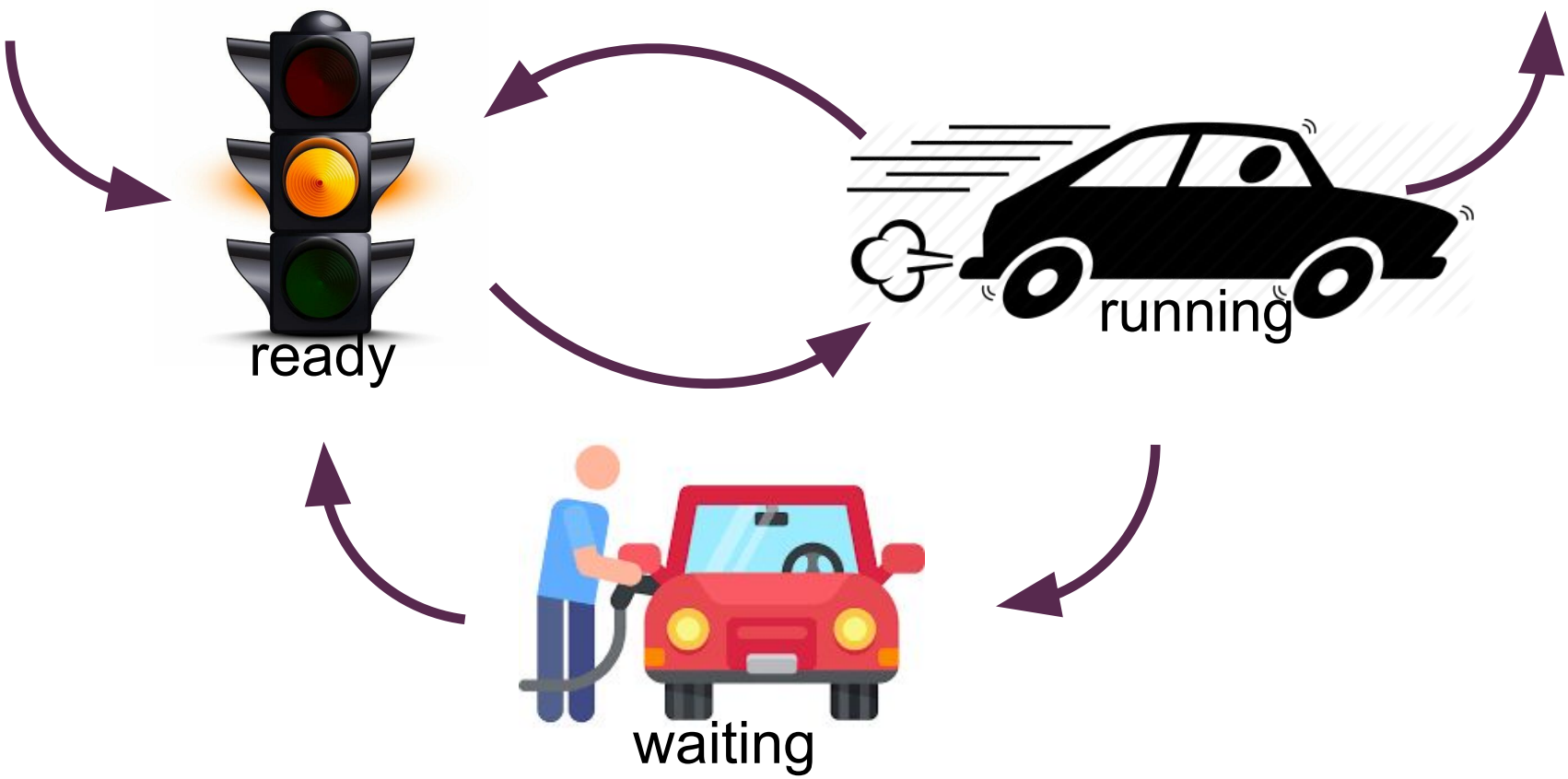
- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur (e.g., I/O completion)
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

שמות המצבים (states) המפורטים בשקף משתנים ממערכת למערכת
כמה תהליכים יכולים להיות במערכת בכל רגע נתון בכל state?

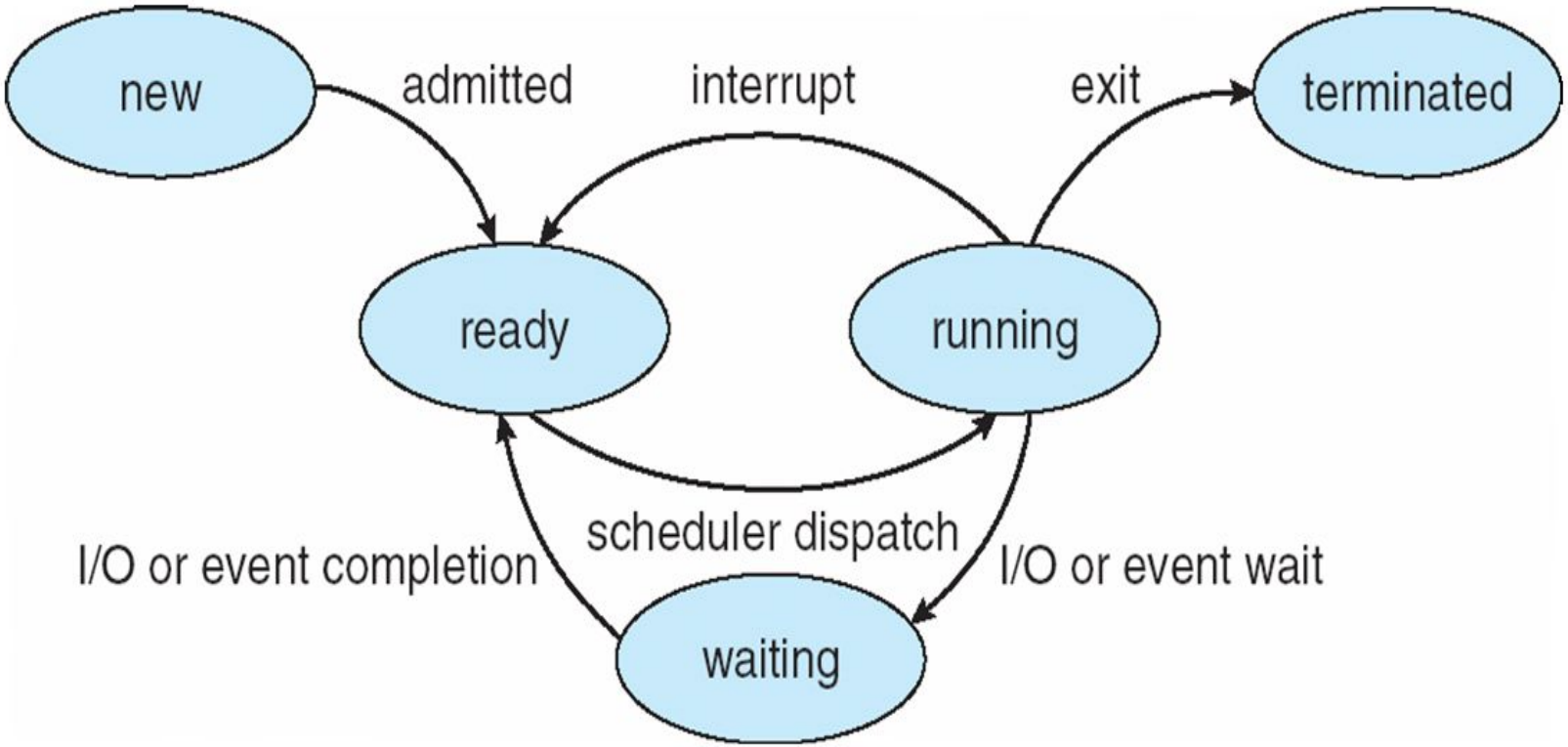
Process State

new

terminated



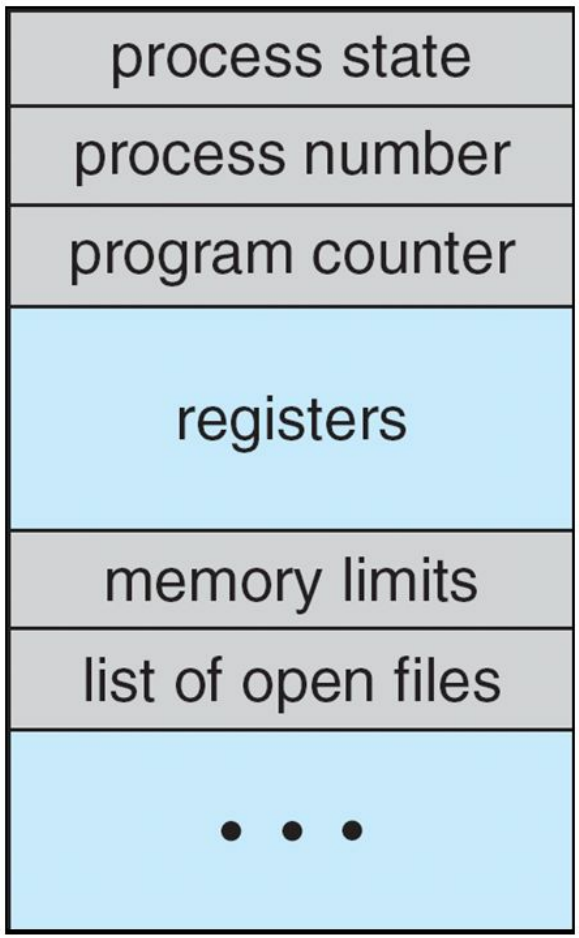
Process State



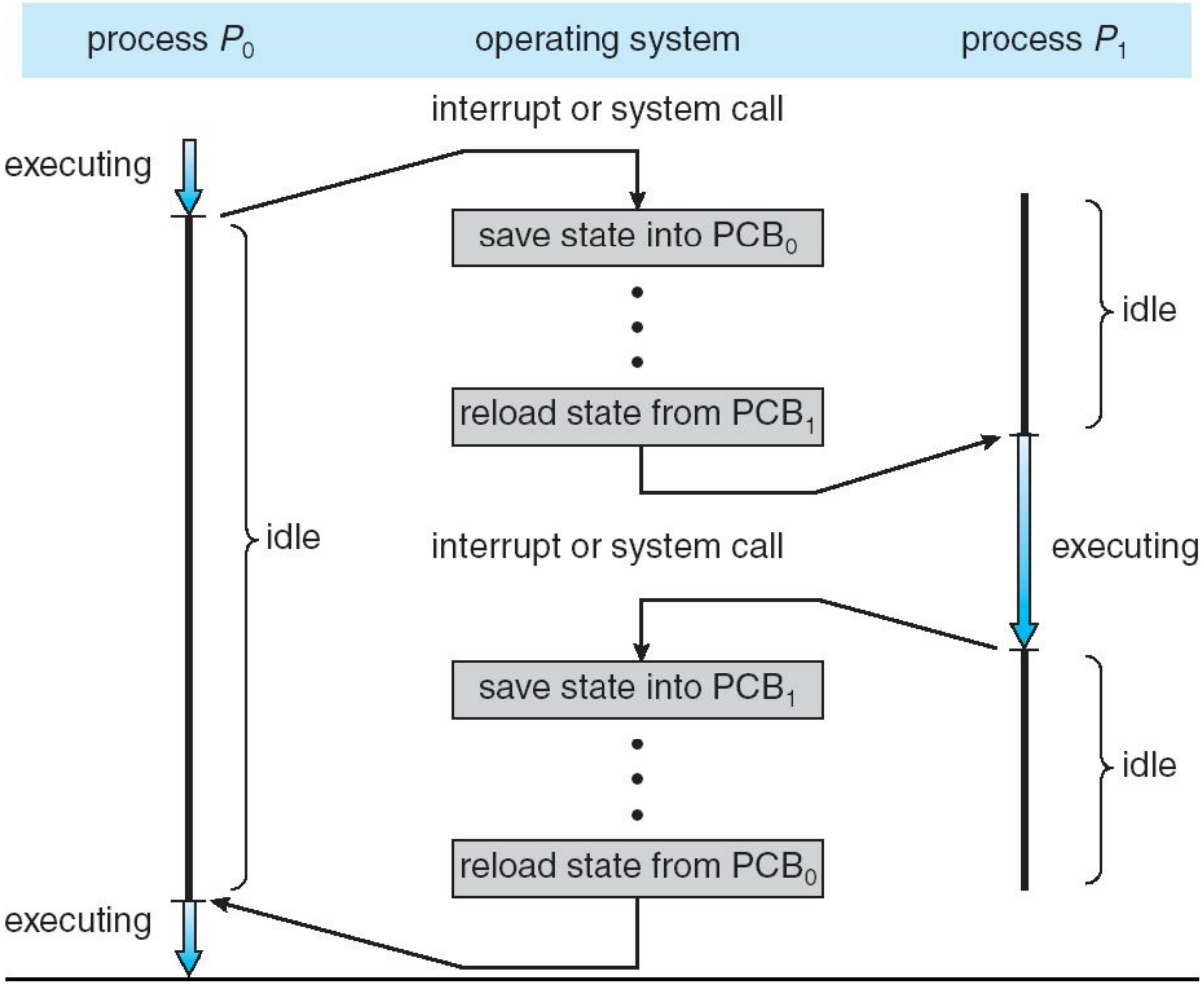
Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter – address of next instruction
- CPU registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information
- I/O status information – I/O devices allocated to process, list of open files



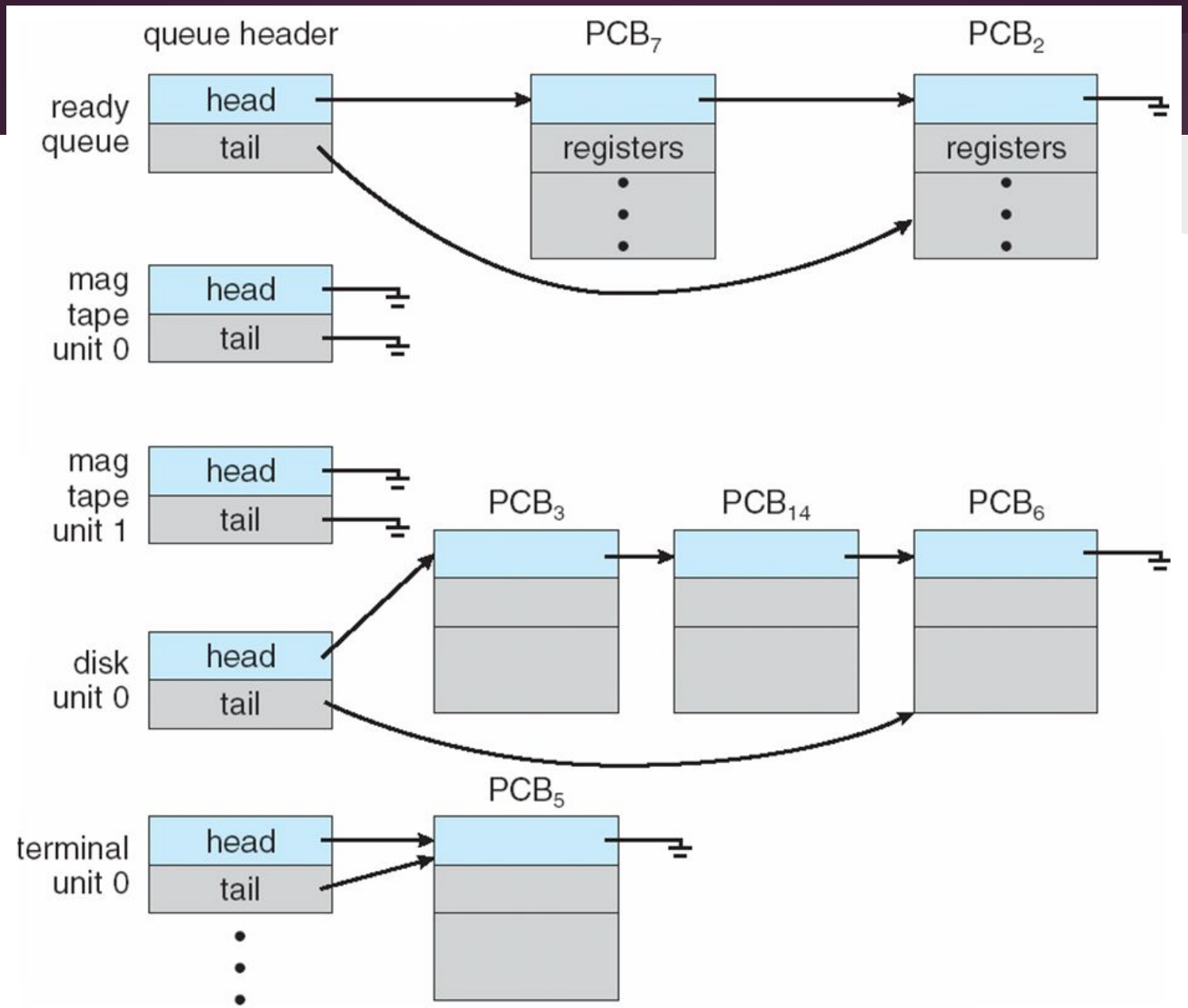
CPU Switch From Process to Process



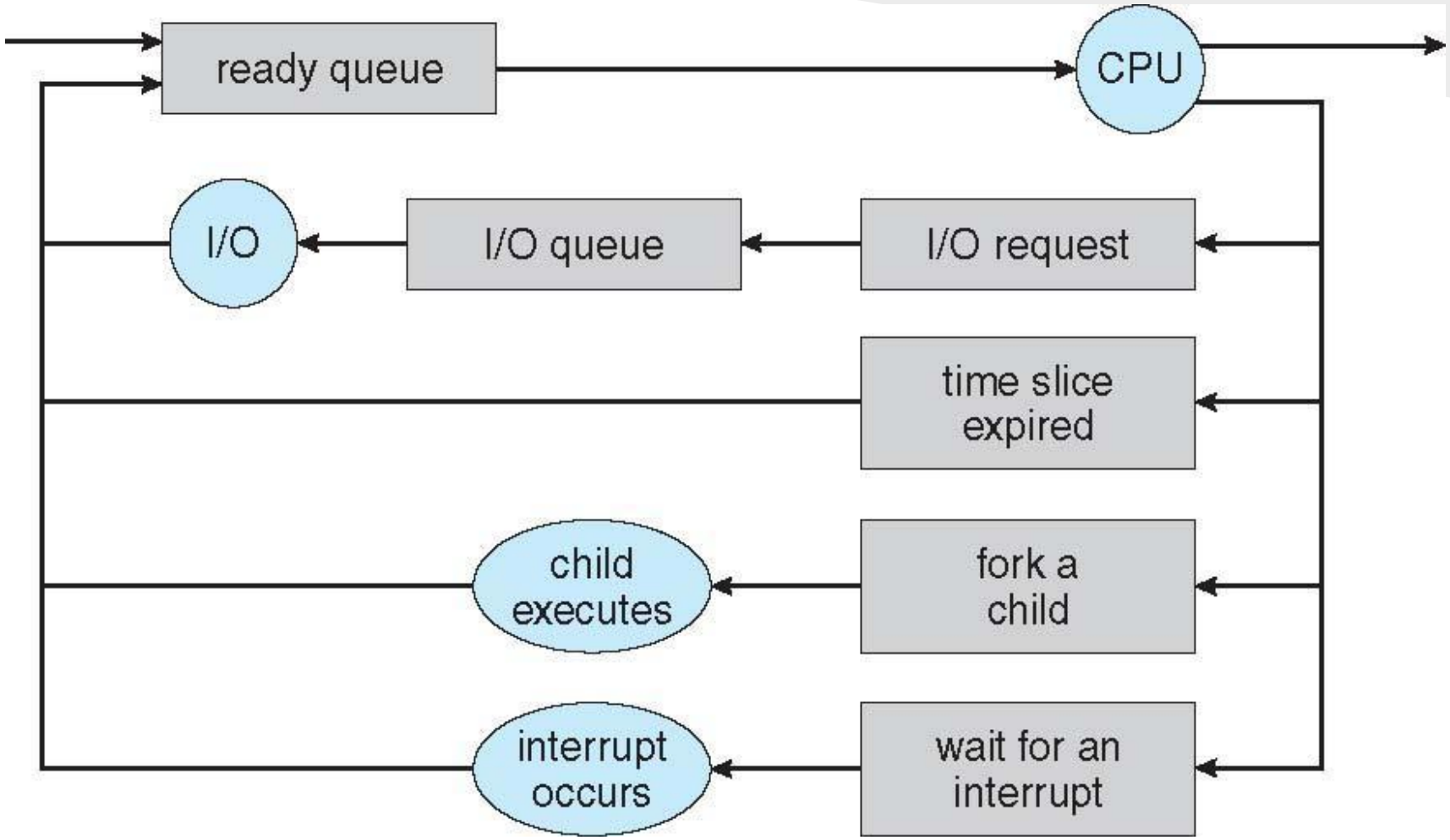
Process Scheduling Queues

- המטרה – שבכל עת יהיה תהליך שיכול לרוץ על המעבד
- למטרה זו נשתמש ב- Process Scheduler
- תורים בהם נעשה שימוש:
  - Job Queue – סט כל התהליכים שבמערכת (לתור זה מגיעים כל התהליכים מיד עם יצירתם)
  - Ready Queue – סט התהליכים אשר:
    - א) נמצאים בזיכרון
    - ב) ממתינים להרצה
  - Device Queue – סט התהליכים הממתינים ל- I/O device
- במהלך הרצתו עובר התהליך בין התורים השונים

Ready Queue And Various I/O Device Queues



Life cycle of a process



Schedulers

ישנם 2 סוגים של Schedulers במ״ה:

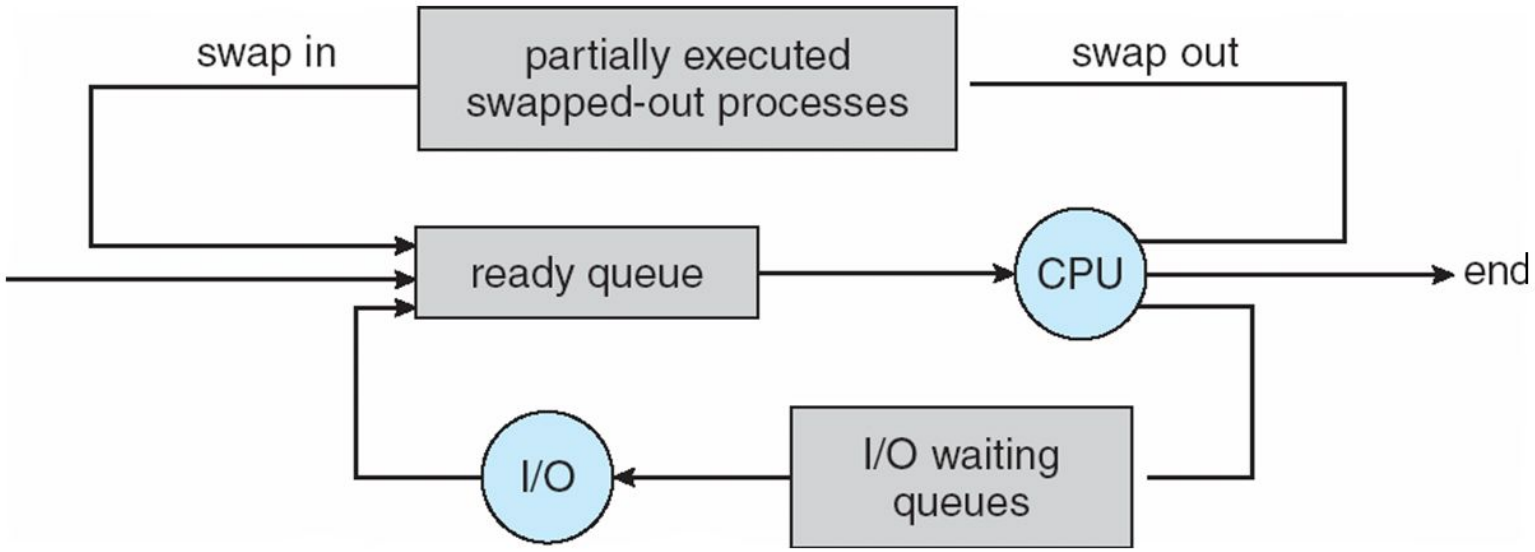
- Short term scheduler
- Long term scheduler

Long-term scheduler

Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue

Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)

The long-term scheduler controls the **degree of multiprogramming** **(number of processes in memory)**



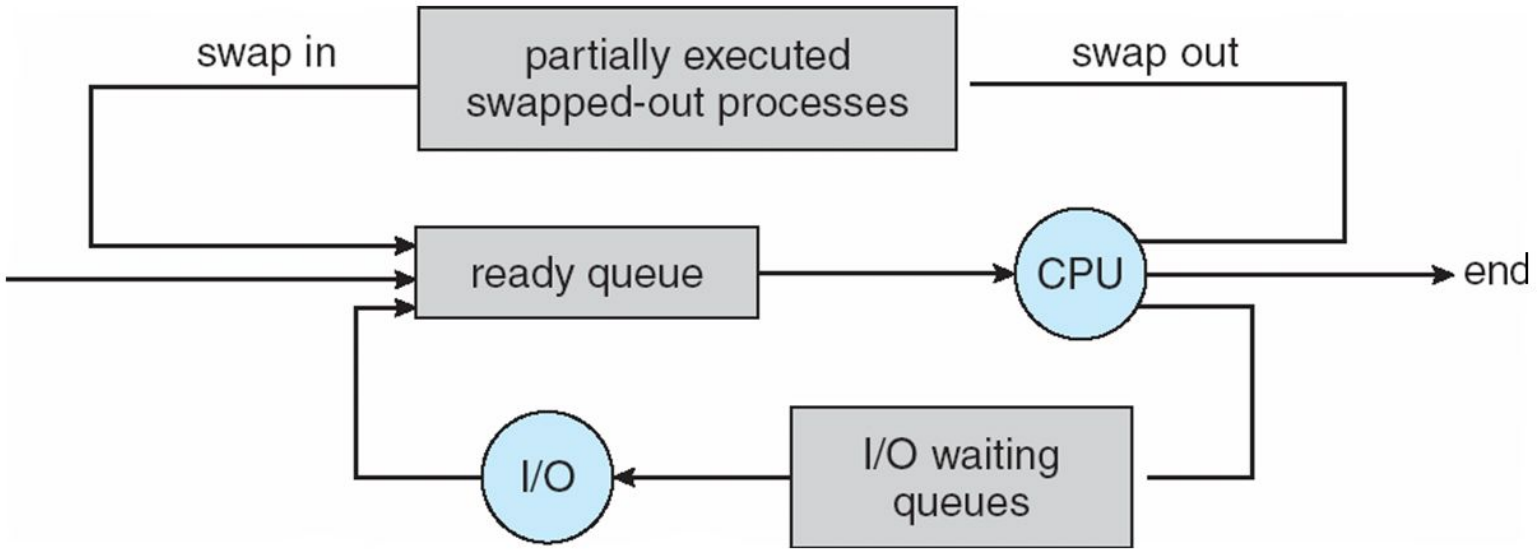
Short-term scheduler

Short-term scheduler (or **CPU scheduler**)

- selects which process should be executed next
- allocates CPU to that process

Sometimes the only scheduler in a system

Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)



Scheduler

Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts

לדוגמה: גלישה ברשת, העתקה של קבצים גדולים...

- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

לדוגמה: חישובים מסובכים

- Long-term scheduler strives for good ***process mix***

מה יקרה ל- ready queue ו- I/O queue כאשר:

- כל התהליכים הם I/O bound?
- כל התהליכים הם CPU bound?

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Context Switch

- כאשר המעבד עובר לטפל בתהליך אחר, המערכת חייבת לשמור את ה- state של התהליך המסיים ו"לטעון" את ה- state השמור של התהליך החדש באמצעות context switch
- ה- context של תהליך מיוצג ע"י ושמור ב- PCB
- זמן ביצוע ה- context switch הוא overhead שכן המערכת איננה מבצעת עבודה שמשמשת את המשתמש
- משך זמן ביצוע ה- context switch מושפע מהתמיכה בחומרה

שאלה?

במערכת מחשב N תהליכים החולקים CPU באמצעות מנגנון RR (N>=2). הנח כי כל content switch לוקח S מילישניות ושכל time quantum אורך Q מילישניות. לצורך פשטות, הנח כי התהליכים אף פעם לא עושים blocking (באף event) ורק עוברים בין ה- CPU וה- ready queue. מהו הערך המקסימלי של Q כך שהזמן המקסימלי בין הרצת שתי instructions של אותו תהליך הוא T מילישניות? (התשובה צריכה להיות פונקציה של N,S,T)

תשובה:

$$Q < \frac{T - N * S}{N - 1}$$

Operations on Processes

System must provide mechanisms for:

- process creation,
- process termination,
- and so on as detailed next

Process Creation

- Creating a process using the create process system call
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options (cpu, mem, i/o)
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

Processes Tree

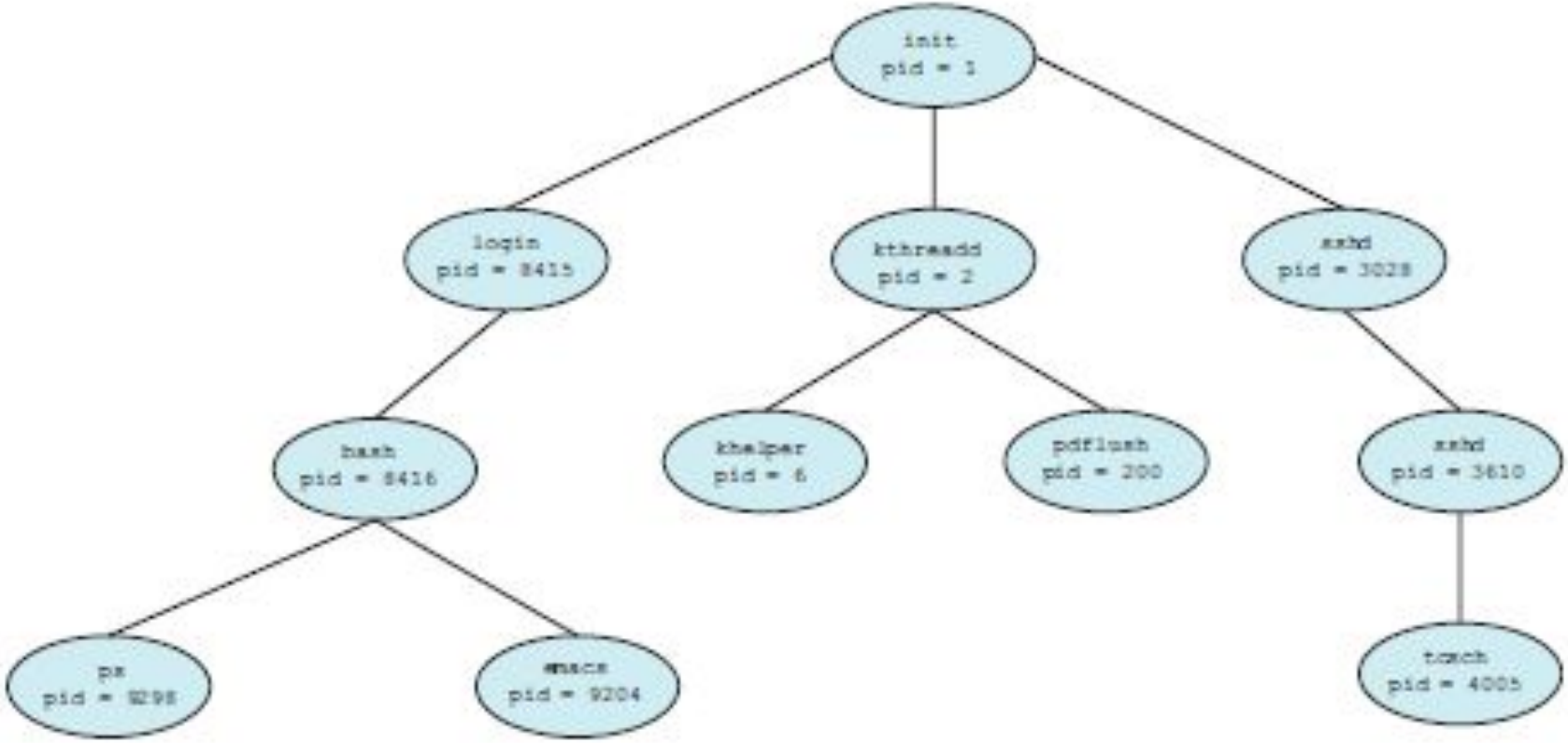


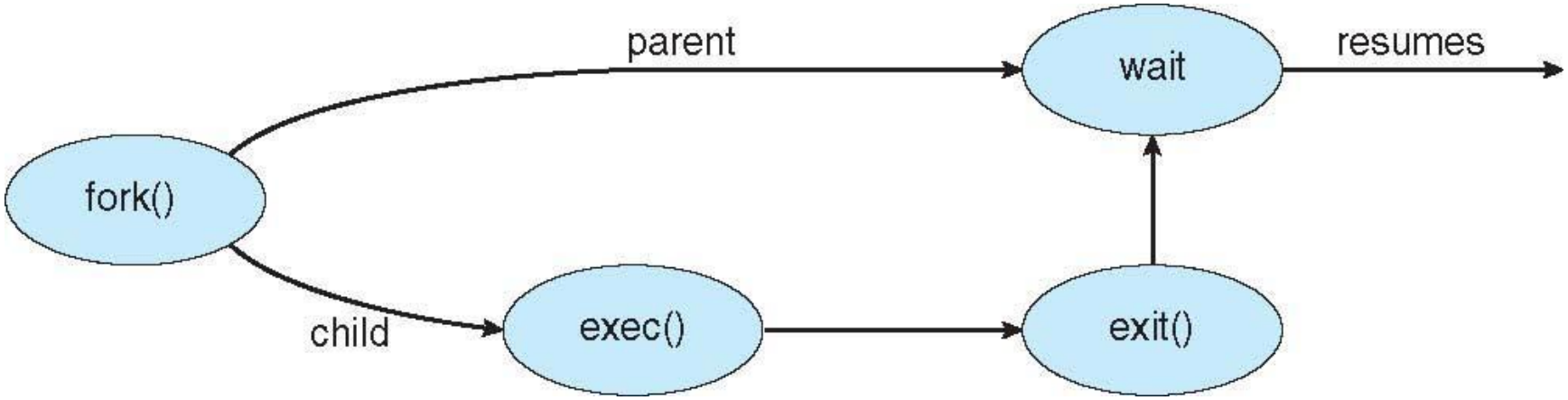
Figure 3.8 A tree of processes on a typical Linux system.

Process Creation

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

Process Creation

- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process' memory space with a new program
  - wait() – takes the process out of ready queue until the child process terminates



C Program Forking Separate Process

```
int main()
{
pid_t  id;
	/* fork another process */
	id = fork();
	if (id < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		exit(-1);
	}
	else if (id == 0) { /* child process */
		execlp("/bin/ls", "ls", NULL);
	}
	else { /* parent process */
		/* parent will wait for the child to complete */
		wait (NULL);
		printf ("Child Complete");
		exit(0);
	}
}
```

Process Termination

בסיום פעולתו נדרש התהליך להריץ את הפקודה exit

- מ"ה מוחקת אותו מרשימת התהליכים
- משאבי התהליך מוחזרים למ"ה לצורך הקצאה מחדש
- אם התהליך הוא תהליך בן אז (במידת הצורך) מועבר output לתהליך האב (via wait)

תהליך אב יכול לגרום להפסקת פעולת תהליך בן (abort) לדוגמה במקרים:

- תהליך הבן משתמש ביותר משאבים ממה שהוקצו לו
- המשימה שלשמה נוצר התהליך הסתיימה או שאינה נדרשת עוד
- תהליך האב עצמו סיים את פעולתו (בחלק ממ"ה לא ניתן להשאיר תהליך רץ כאשר תהליך האב מסתיים, במקרה כזה עושים cascade termination)

תהליך זומבי

תהליך זומבי במערכות הפעלה מבוססות Unix הוא תהליך שביצועו הושלם, אך עדיין נכלל בטבלת התהליכים של מערכת ההפעלה.
הרשומה שנותרה בטבלת התהליכים מאפשרת לתהליך האב שיצר את התהליך שסיים את ריצתו (ועכשיו הוא זומבי) לקרוא את ערך היציאה של הזומבי.

exit vs return

מה ההבדל בין exit לבין return?

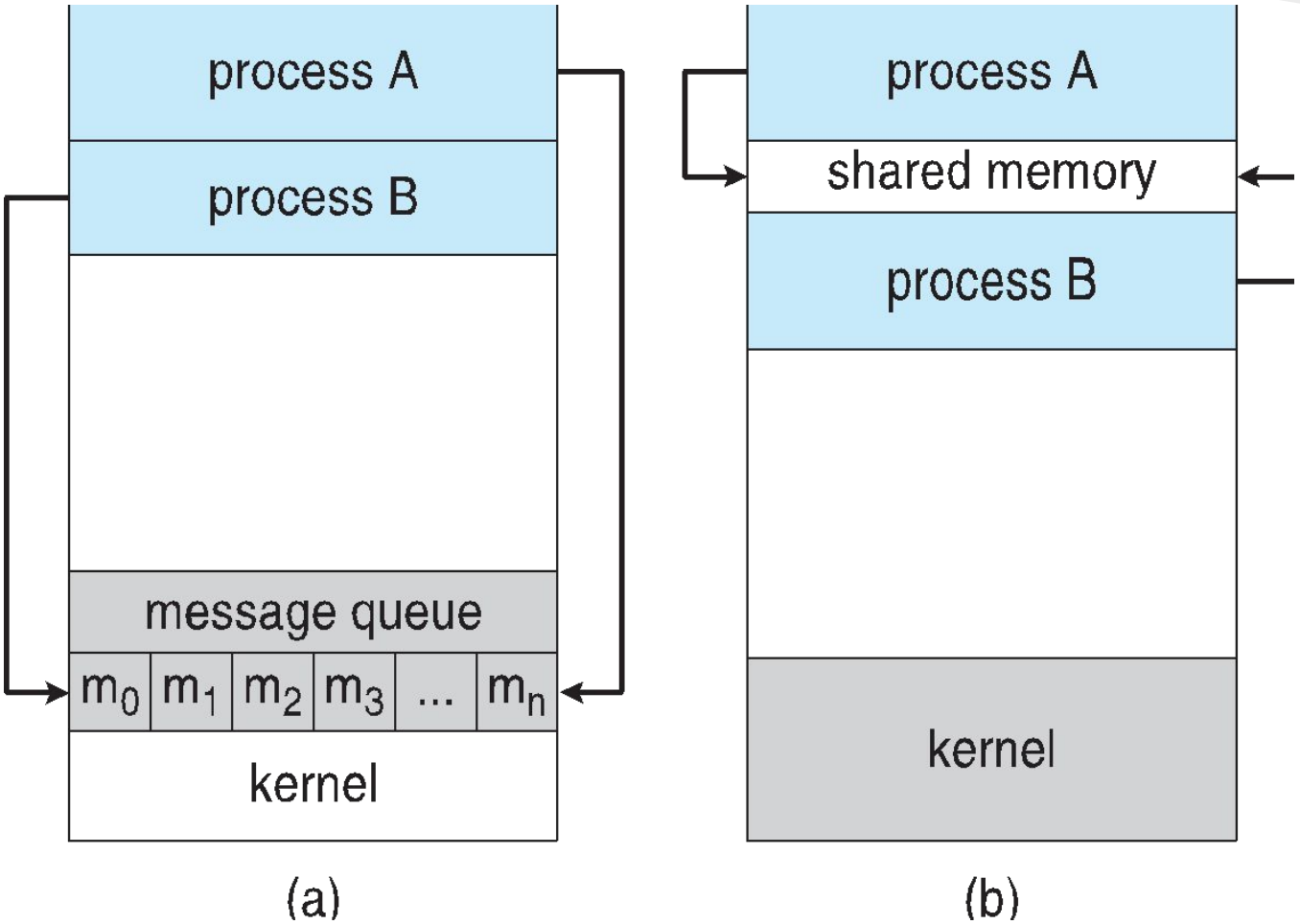
- **return** is an instruction of the language that returns from a function call.
- **exit** is a system call (not a language statement) that terminates the current process.

Interprocess Communication

- תהליכים צריכים דרך לחלוק ולשתף מידע בניהם:
- בין שרצים באותה מערכת ובין שרצים במערכות שונות
- התמיכה בשיתוף מידע היא באמצעות IPC
- שני מודלים של IPC:
  - Shared memory
  - Message passing

Communications Models

(a) Message passing. (b) shared memory.



Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information
  - *consumer* process consume information
- Shared memory solution:
  - **unbounded-buffer** no practical limit on the size of the buffer

באפר ללא הגבלה, ניתן לכתוב כמה שרוצים ללא תלות בקורא

  - **bounded-buffer** assumes that there is a fixed buffer size

באפר בגודל סופי לכן יש מגבלה על כמות המידע שניתן לכתוב לפני שהמקום מתפנה ע"י הקורא

Bounded-Buffer – Shared-Memory Solution

Shared data

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

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

פוינטר לסלוט במערך אליו צריך להכניס את האיבר הבא

פוינטר לסלוט אותו יש לקרוא

המערך מעגלי,

כאשר in==out המערך ריק,

כאשר in+1==out המערך מלא

Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

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

IPC – Message Passing

- מהווה מנגנון לתקשורת וסנכרון בין תהליכים
- הרעיון הוא שתהליכים יתקשרו ללא צורך בגישה למשתנים משותפים
- ה- IPC-facility מספק שתי פעולות:
  - send(message)
  - receive(message)
- אם P ו- Q רוצים לתקשר ביניהם עליהם:
  - לייצר communication link ביניהם
  - להעביר הודעות באמצעות send/receive
- מימוש ה- communication link הוא באמצעות חומרה (shared hardware bus, memory וכו')

Message Passing

- Direct – ההודעה מועברת בצורה ישירה מתהליך אחד לתהליך השני באמצעות הפקודות send/receive
- Indirect – ההודעה מועברת לצד שלישי ולא בצורה ישירה לדוגמה באמצעות mailbox (יכול להיות משותף למספר תהליכים, זוג תהליכים יכול להשתמש יותר מ- mailbox אחד)

Direct- process must name each other explicitly:
send(P, message) send a message to process P
receive(Q, message) receive a message from process Q

Indirect – must define a common mailbox A:
send(A, message) – send a message to mailbox A
receive(A,message) – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or nonblocking
- blocking = synchronous
  - The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - The receiver blocks until a message is available.
- nonblocking = asynchronous
- The sending process sends the message and resumes operation.
- Nonblocking receive. The receiver retrieves either a valid message or a null.

Buffering

Messages exchanged by communicating processes reside in a temporary queue

Such queues can be implemented in three ways:

- **Zero capacity** - queue maximum length of zero -> the sender must block until the recipient receives the message.
- **Bounded capacity -** queue has finite length n -> if Q not full the sender can continue execution after sending the message
- **Unbounded capacity -** queue's length is infinite -> The sender never blocks.

IPC - examples

- POSIX shared memory
- Pipes

shared memory - producer

```
/* create the shared memory object */
fd = shm_open(name, O_CREAT | O_RDWR,0666);
/* configure the size of the shared memory object */
ftruncate(fd, SIZE);
/* memory map the shared memory object */
ptr = (char *)
mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
/* write to the shared memory object */
sprintf(ptr,"%s",message 0);
```

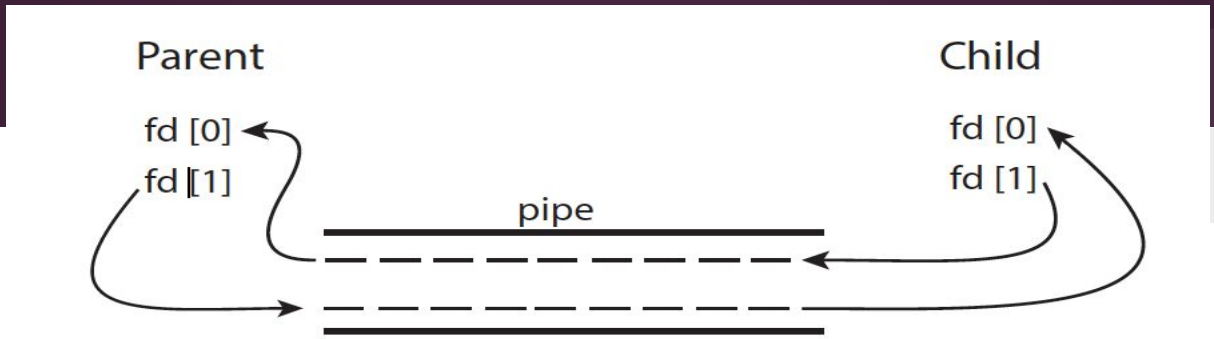
shared memory - consumer

```
/* open the shared memory object */
fd = shm_open(name, O_RDONLY, 0666);
/* memory map the shared memory object */
ptr = (char *)
mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
/* read from the shared memory object */
printf("%s",(char *)ptr);
/* remove the shared memory object */
shm_unlink(name);
```

int pipe(int *pipefd*[2]);

- **pipe**() creates a pipe, a unidirectional data channel that can be used for interprocess communication.
- The array *pipefd* is used to return two file descriptors referring to the ends of the pipe.
- *pipefd[0]* refers to the read end of the pipe.
- *pipefd[1]* refers to the write end of the pipe.
- Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

Pipes



- Does the pipe allow bidirectional communication, or is communication unidirectional?
  - half duplex - data can travel only one way at a time
  - full duplex - data can travel in both directions at the same time
- Must a relationship (such as parent–child) exist between the communicating processes?
- Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

Pipes

```
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* create the pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
```

Pipes - sender

```
if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);
    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
    /* close the write end of the pipe */
    close(fd[WRITE_END]);
}
```

Pipes - reciever

```
else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);
    /* read from the pipe */
    read(fd[READ END], read_msg, BUFFER_SIZE);
    printf("read %s",read msg);
    /* close the read end of the pipe */
    close(fd[READ_END]);
}
```

Communication in client-server systems

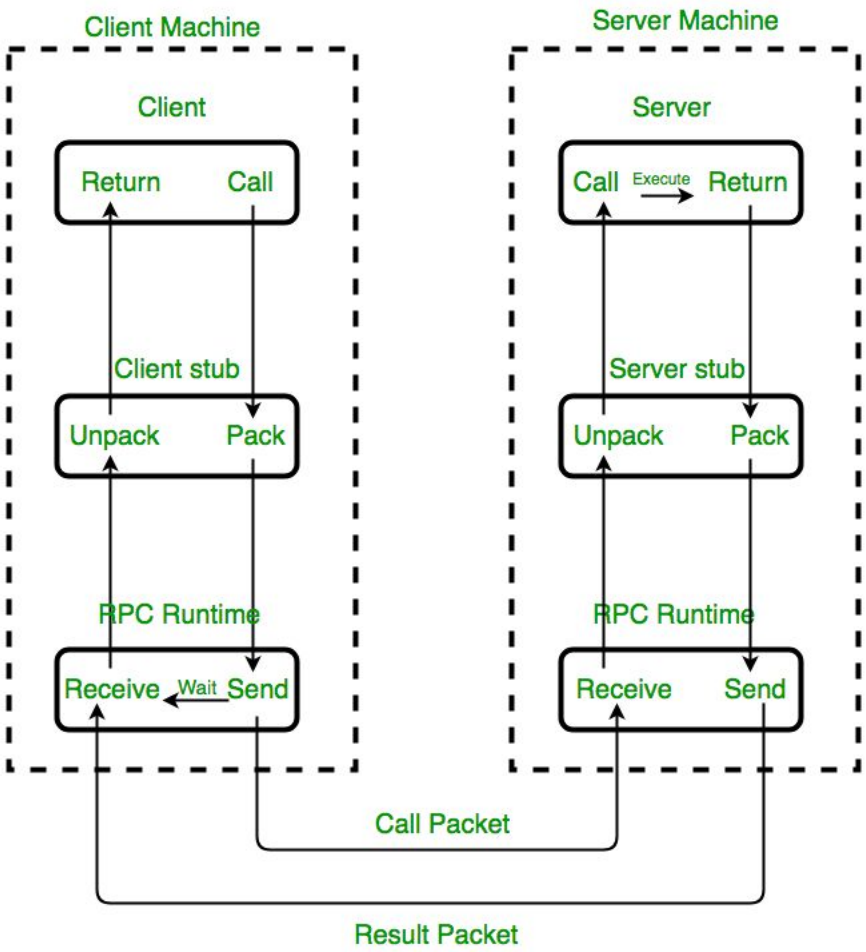
- Sockets
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)

Sockets

- socket is defined as an endpoint for communication
- A socket is identified by an IP address and a port number (and protocol)
- Protocols TCP/UDP
- sockets use a client–server architecture

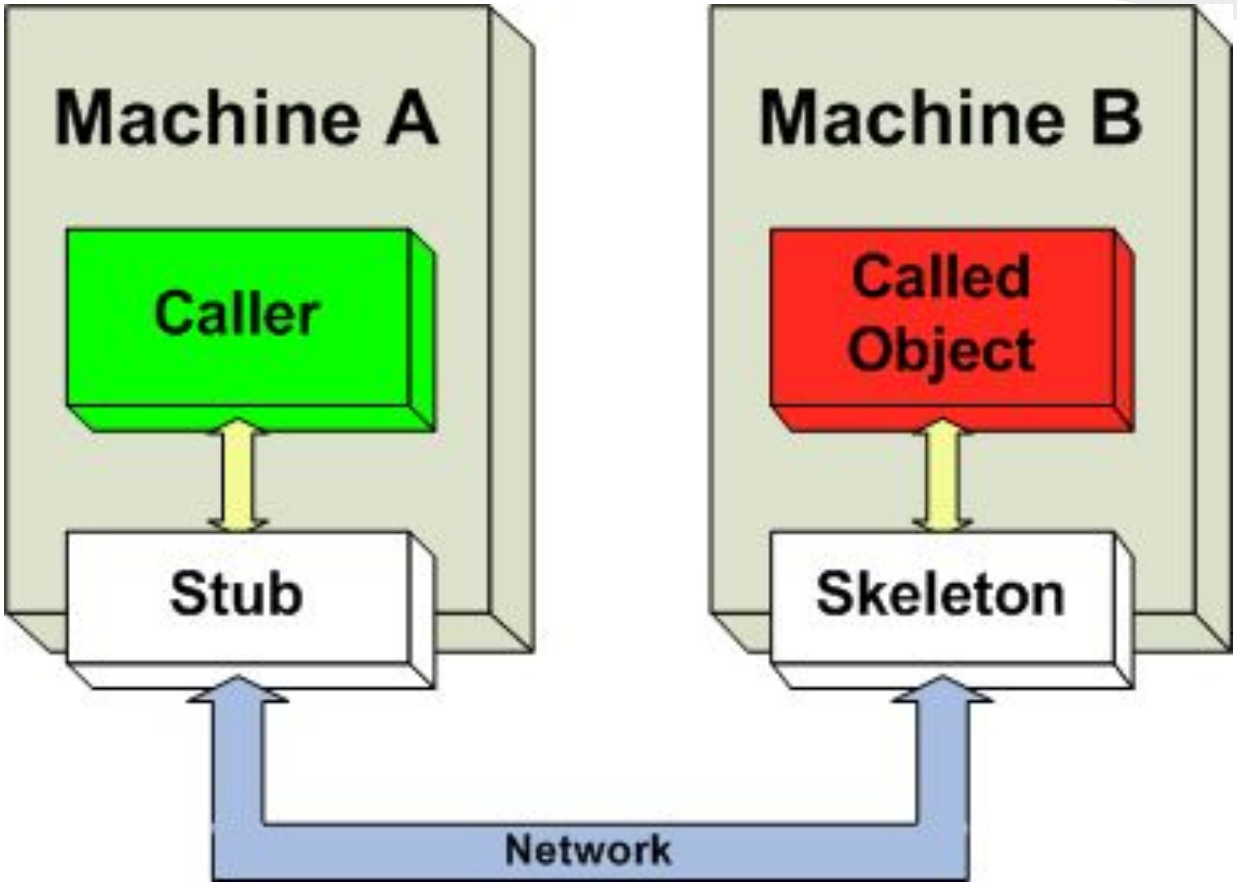
RPC

- Abstracts procedure calls between processes on networked systems
- Stub – client side proxy for the actual procedure on the server



Implementation of RPC mechanism

RMI


Machine A
Caller
Stub
Machine B
Called Object
Skeleton
Network

שאלה?

האם יש הגיון לבצע busy waiting כאשר עובדים עם מעבד יחיד?

לא, כי במקום להעסיק את המעבד בהמתנה פעילה אפשר לפנות אותו לקדם תהליכים אחרים עד שהאירוע שחיכינו לו יקרה.

שאלה?

```
8
9  #import <Foundation/Foundation.h>
10 #include <sys/types.h>
11 #include <stdio.h>
12 #include <unistd.h>
13 int value = 5;
14 int main()
15 {
16     pid_t pid;
17     pid = fork();
18     if (pid == 0) { /* child process */
19         value += 15;
20         return 0;
21     }
22     else if (pid > 0) { /* parent process */
23         wait(NULL);
24         printf("PARENT: value = %d",value); /* LINE A */
25         return 0;
26     }
27 }
28
```

מה יהיה יודפס על המסך עבור תוכנית זו?

תשובה

```
PARENT: value = 5
Program ended with exit code: 0
```

לכל תהליך יש את מרחב הזיכרון שלו,
לכן הם לא חולקים את המשתנים

שאלה

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main()
4  {
5      /* fork a child process */
6      fork();
7      /* fork another child process */
8      fork();
9      /* and fork another */
10     fork();
11     return 0;
12 }
13
```

כמה תהליכים ייווצרו בתוכנית הבאה כולל התהליך הראשי של התוכנית?

תשובה

8 תהליכים!

שאלה

The Sun UltraSPARC processor has multiple register sets for context switch, a special register “current-register-set” point to the active register set.

- Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets.
- What happens if the new context is in memory rather than in a register set and all the register sets are in use?

תשובה

- The CPU current-register-set pointer is changed to point to the set containing the new context, which takes very little time.
- If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. Then the current-register-set pointer is changed to point to the set containing the new context.

This operation takes more time…