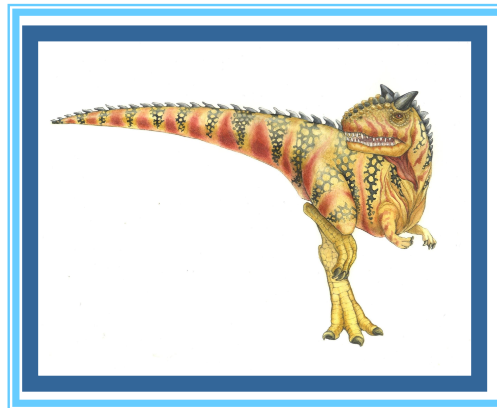
פרק 3 – תהליכים (Processes)



Revised and updated by David Sarne

שאלה 2

א. למה משמש ה-CLI?

ב. מה היתרון העיקרי של ה-CLI על-פני ה-GUI?

ג. מה תפקידו של ה-interpreter ומה הן שתי השיטות העיקריות למימושו? מהו החיסרון העיקרי של כל אחת משתי השיטות.

<u>תשובות</u>

א. ה- (Command Line Interpreter) CLI מאפשר הרצה של פקודות באמצעות הזנה ישירה על-ידי המשתמש.

ב. היתרון הוא שבאמצעות פקודה אחת ניתן לבצע פעולה מורכבת שהיו נדרשים בשבילה הרבה מאוד "קליקים" ב- GUI.

ג. ה- interpreter מתרגם את פקודות המשתמש ל- system calls.

a. שיטת מימוש 1: כחלק מהקרנל (חיסרון: מחייב שינוי של ה- SHELL כאשר רוצים לשנות/להוסיף פקודות).

b. שיטת מימוש 2 – באמצעות קבצי תוכניות אותן הוא מריץ (חיסרון: איטי יותר).

Introduction

- מערכות המחשב (ומערכות ההפעלה) הראשונות:
  - אפשרו הרצת תכנית אחת (בלבד) בו זמנית
    - לתכנית המורצת היתה שליטה מלאה על המערכת והמשאבים
- כיום:
  - מאפשרות העלאה לזיכרון והרצה של מספר תכניות לזיכרון והרצה במקביל
    - מחייב שליטה הדוקה יותר של מערכת ההפעלה

מטרות השיעור

- להציג את ה"תהליך" (process) – תוכנית בהרצה (program in execution),
- לתאר מספר מאפיינים (features) של תהליכים, ובכלל זה תזמון (scheduling), יצירה (creation), עצירה (termination) ותקשורת (communication)

Process Concept

- בקורס נשתמש במושגים job ו- process כחליפיים
  - Jobs – בד"כ מתייחס ל- batch systems
  - Process – בד"כ למערכות שהן time-shared
- התהליכים הרצים במערכת:
  - תהליכים של מערכת ההפעלה המריצים system code
  - תהליכים של המשתמש המריצים user code
- Process – "תוכנית בהרצה":
  - Program (executable) – ישות פאסיבית השוכנת על הדיסק
  - Process – ישות אקטיבית אשר משנה באופן רצוף את מצבה
- הרצת התהליך מתבצעת בצורה סדרתית (sequential)
- התהליך כולל:
  - program counter
  - Stack and heap
  - Text and data section

Process in Memory

max

stack

heap

data

text

0

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
}
```

Galvin and Gagne ©2009

Program and Process

- על-בסיס אותה תוכנית ניתן להריץ שני תהליכים או יותר (במקביל), כשלכל אחד יש execution sequence משלו

Process State

- במהלך ריצתו משנה התהליך 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?

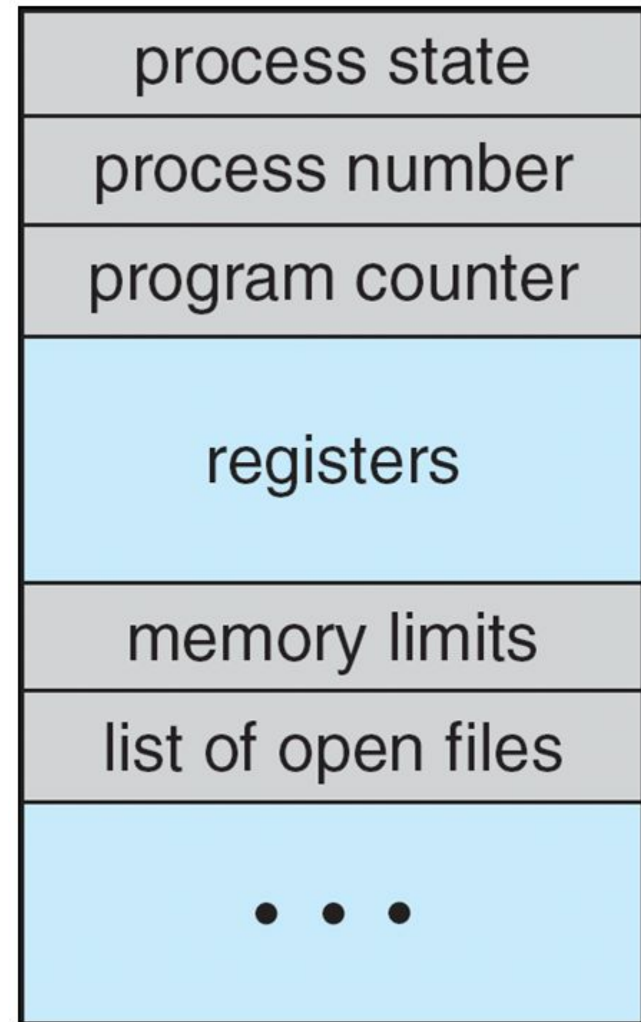
Diagram of Process State

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter – address of next instruction
- CPU registers
- CPU scheduling information – e.g., process priority, pointers to scheduling queues
- Memory-management information – value of base and limit registers, page or segment tables
- Accounting information
- I/O status information – list of open files, I/O devices allocated to the process



CPU Switch From Process to Process

process P_0 | operating system | process P_1

interrupt or system call

executing

save state into PCB_0

•
•
•

reload state from PCB_1

idle

idle

interrupt or system call

executing

save state into PCB_1

•
•
•

reload state from PCB_0

idle

executing

Process Scheduling Queues

- המטרה – שבכל עת יהיה תהליך שיכול לרוץ על המעבד (לצורך מקסום ניצולת 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 כל התהליכים אשר: א) נמצאים בזיכרון; ב) ממתינים להרצה
  - Device queues – סט התהליכים הממתינים ל- I/O device
- במהלך הרצתו עובר התהליך בין התורים השונים

Ready Queue And Various I/O Device Queues

queue header

PCB$_7$

PCB$_2$

ready queue

head

tail

registers

registers

mag tape unit 0

head

tail

mag tape unit 1

head

tail

PCB$_3$

PCB$_{14}$

PCB$_6$

disk unit 0

head

tail

PCB$_5$

terminal unit 0

head

tail

Representation of Process Scheduling

Queuing Diagram

Why start at ready queue?

ready queue

CPU

I/O

I/O queue

I/O request

time slice expired

Wait for the child to finish execution

child executes

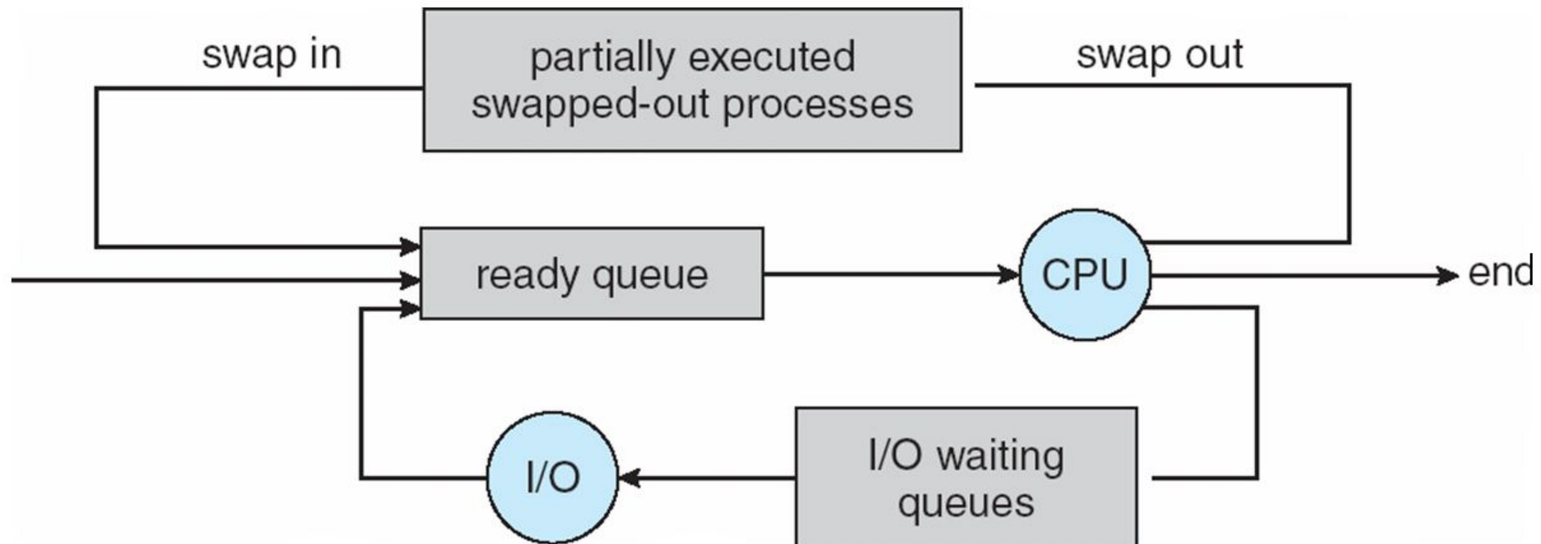
fork a child

interrupt occurs

wait for an interrupt

Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue – loads processes from disk to memory
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next (from the ready queue) and allocates CPU



Schedulers (Cont)

- ה- short-term scheduler נכנס לפעולה בצורה תדירה (כל מספר מילישניות):
  - חייב להיות מהיר, אחרת יצור overhead משמעותי
  - עושה שימוש בשיטות יחסית פשוטות כמו FCFS, priority scheduling
- ה- long term scheduler:
  - מופעל בתדירות נמוכה יותר (כל כמה שניות או דקות) ולכן פעולתו יכולה להיות ארוכה יותר (מנגנונים יותר מתוחכמים)
  - שולט על ה- degree of multiprogramming (מספר התהליכים בזיכרון)

Schedulers (Cont)

- תהליכים מאופיינים כ:
  - I/O bound – מבלים את רוב זמנם בפעולות I/O ומעט מאוד בפעולות חישוביות (יחסית קצרות – למשל גלישה ברשת, העתקה של קבצים גדולים)
  - CPU bound – מבלים את רוב זמנם בתהליכים חישוביים (ארוכים ומעטים) – למשל חישוב פאי (אם יקבלו מעבד פי שתיים יותר מהיר יסיימו במחצית מהזמן)
- ה- long term scheduler צריך לדאוג שבכל רגע יהיה בזיכרון מיקס של תהליכי I/O-bound ו- CPU-bound:
  - מה יקרה ל- ready queue ו- I/O queues כאשר:
    - כל התהליכים הם I/O bound?
    - כל התהליכים הם CPU-bound?

Context Switch

- כאשר המעבד עובר לטפל בתהליך אחר, המערכת חייבת לשמור את ה- state של התהליך המסיים ו"לטעון? את ה-stateהשמור של התהליך החדש באמצעות context switch
- ה- context של תהליך מיוצג על-ידי 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:
- E.g., Sun UltraSPARC provides multiple sets of registers

<u>שאלה 3</u> (10 נקודות)

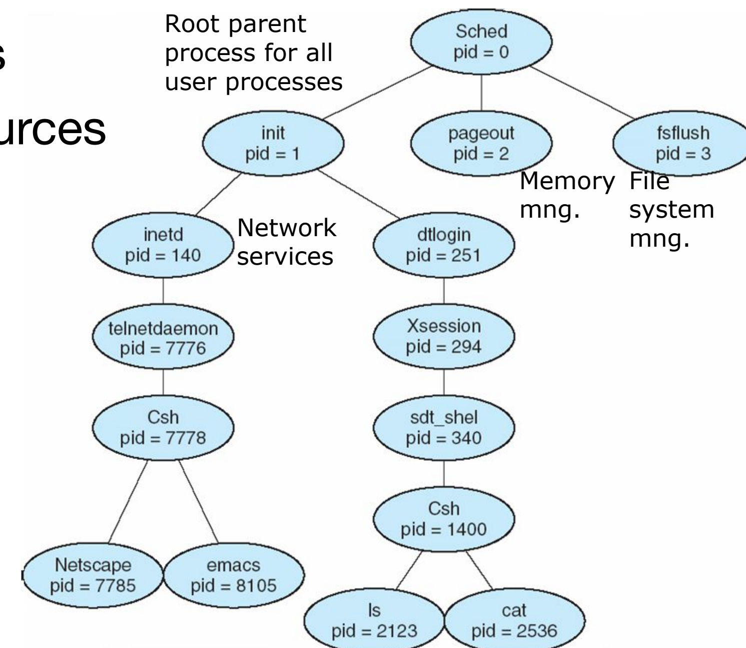
במערכת מחשב N תהליכים החולקים CPU באמצעות מנגנון RR (N>=2). הנח כי כל context switch לוקח S מילישניות ושכל time quantum אורך Q מילישניות. לצורך פשטות, הנח כי התהליכים אף פעם לא עושים blocking (באף event) ורק עוברים בין ה-CPU וה- ready queue. מהו הערך המקסימלי של Q כך שהזמן המקסימלי בין הרצת שתי instructions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

תשובה:

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

Process Creation

- Creating a process – using 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 (cpu time, memory, I/O devices)
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
    - One process can overload the system by creating many processes
- Initialization data is passed from parent to child upon creation

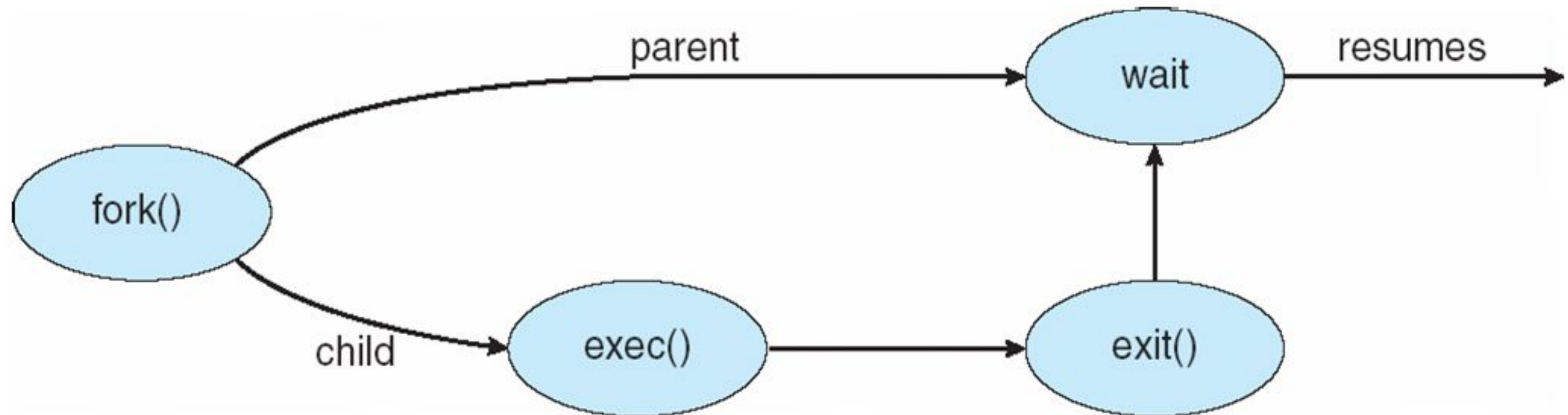


A tree of processes on a typical Solaris

Process Creation (Cont)

- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent – same program and data
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
    - Parent's address space is duplicated
    - Both processes resume execution at the instruction after the fork()
  - **exec** system call used after a **fork** to replace the process' memory space with a new program
  - Wait() – takes the process out the ready queue until the child process terminates

Process Creation

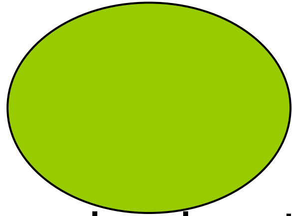


C Program Forking Separate Process

```
int main()
{
pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
        }
        else if (pid == 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:



  - מערכת ההפעלה מוחקת אותו מרשמית התהליכים
  - משאבי התהליך מוחזרים למ"ה לצורך הקצאה מחדש
  - אם התהליך הוא תהליך בן אז (במידת הצורך) output מועבר 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 תהליך הבן הסתיימה או שאינה נדרשת עוד
  - תהליך האב עצמו סיים את פעולתו:
    - בחלק ממערכות ההפעלה לא ניתן להשאיר תהליך רץ כאשר תהליך האב מסתיים – במקרה כזה עושים cascading termination

Interprocess Communication

- תהליכים צריכים דרך לחלוק ולשתף מידע ביניהם:
  - בין שרצים באותה מערכת ובין שרצים במערכות שונות
- התמיכה בשיתוף מידע היא באמצעות IPC (interprocess communication)
- שני מודלים של IPC:
  - Shared memory – usually resides in the address space of creating process
  - Message passing

Communications Models

process A M
process B M
kernel M
2 1
(a)

process A
shared
process B
kernel
1 2
(b)

- Better for exchanging small messages (no conflicts need to be avoided)
- Easier to implement

- Faster

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process (e.g., compiler produces assembly code, consumed by the assembler)
- *Shared memory solution:*
  - *unbounded-buffer* places no practical limit on the size of the buffer (producer never waits)
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
while (true) {
     while (in == out)
            ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
return item;
}
```

Interprocess Communication – Message Passing

- מהווה מכאניזם לתקשורת וסנכרון בין תהליכים
- הרעיון הוא שהתהליכים יתקשרו ללא צורך בגישה למשתנים משותפים (shared variables)
- 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 פקודות send/receive
- מימוש ה- communication link הוא באמצעות חומרה (shared memory, hardware bus וכו')

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
- Main disadvantage:
  - Changing the identifier of a process result with many changes in the communication scheme

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was
    - Alternatively – define an algorithm for selecting which process will receive the message (e.g., round robin)

Synchronization

- מנגנון ה- message passing יכול להיות מאופיין כ- blocking or non-blocking
- תצורת blocking נחשבת תצורה סינכרונית (synchronous):
  - Blocking send – השולח מבצע block עד שההודעה מתקבלת
  - Blocking receive – המקבל מבצע block עד שההודעה זמינה לקבלה
- תצורת non-blocking נחשבת תצורה א-סינכרונית:
  - Non-blocking send – השולח שולח את ההודעה וממשיך בפעולתו
  - Non-blocking receive – המקבל מקבל את ההודעה גם אם היא null או invalid

Buffering

- Whether direct or indirect, messages must reside in a queue; implemented in one of three ways:
  1. Zero capacity – no messages can wait in queue. Sender must block until recipient receives the message
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Communications in Client-Server Systems

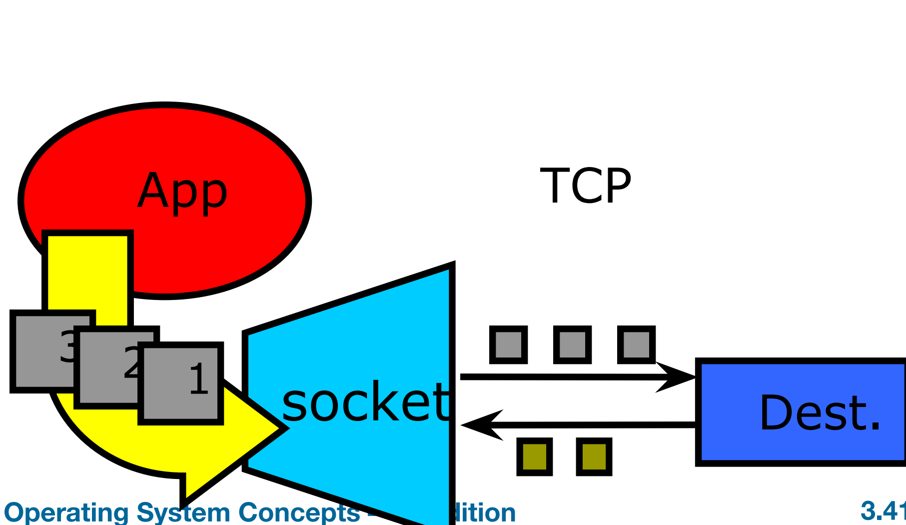
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

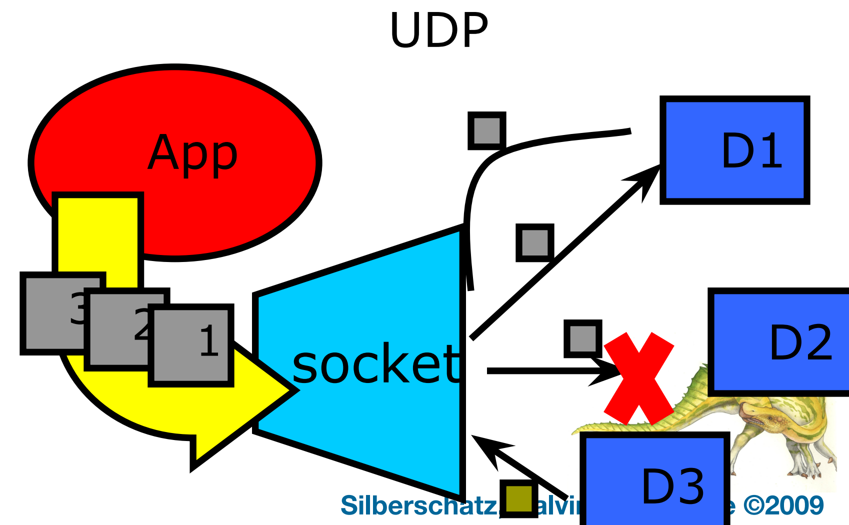
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

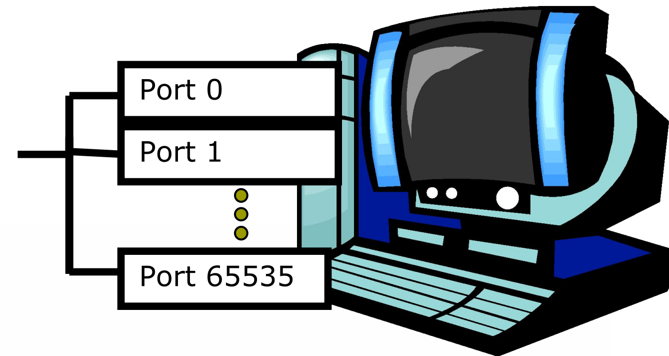
What is a socket?

- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented (e.g., phone) vs. connectionless (e.g., mail)
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)





Socket Communication

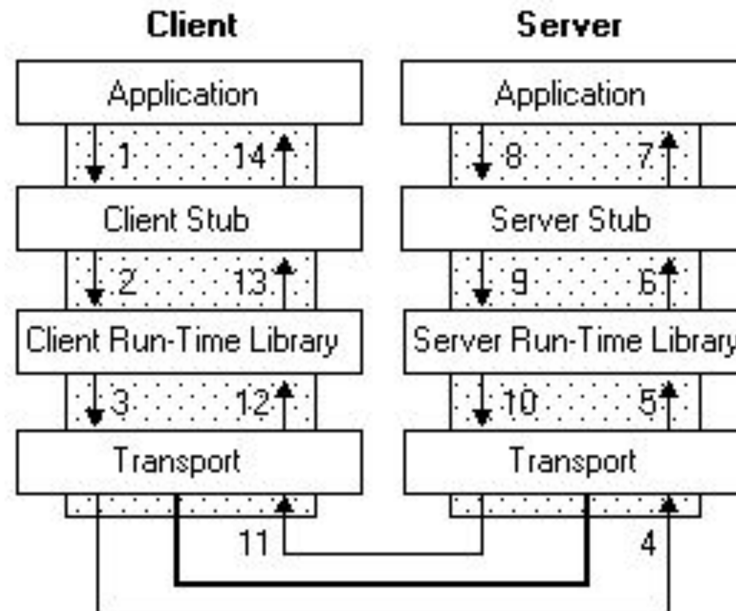


Server waits for incoming client requests by listening to a specified port

Servers implementing specific services (such as telnet, FTP, and HTTP) listen to well-known ports (a telnet server listens to port 23; an FTP server listens to port 21; and a Web, or HTTP, server listens to port 80)

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object

JVM
Java program
remote method invocation
JVM
remote object

Marshalling Parameters

Client Program

Server Program

RMI System

Stubs & Skeletons

Remote Reference Layer

Stubs & Skeletons

Remote Reference Layer

Transport Layer

intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service

understands how to interpret and manage references made from clients to the remote service objects.

client

val = server.someMethod(A,B)

stub

remote object

boolean someMethod (Object x, Object y)

{

implementation of someMethod

. . .

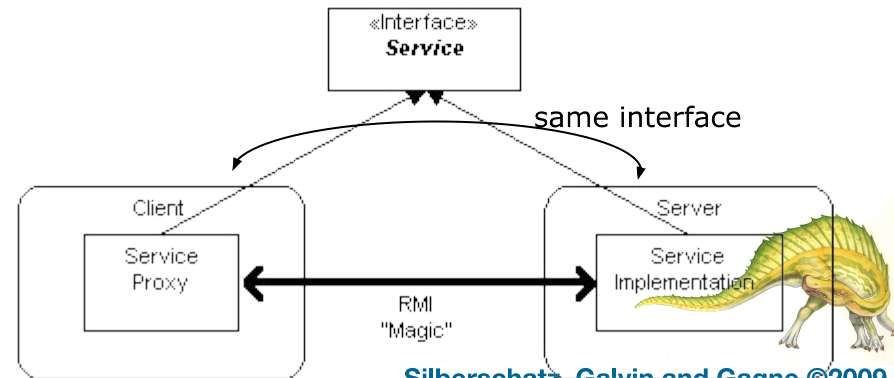
}

skeleton

A, B, someMethod

boolean return value

The RMI principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.



שאלה 2 (6 נקודות)

האם יש היגיון לבצע busy waiting כאשר עובדים עם מעבד יחיד (uniprocessor)? כן / לא (הקף בעיגול). במידה ותשובתך כן, הבא דוגמה. במידה ולא, נמק.

תשובה

לא. אין בכך כל היגיון משום שניתן להריץ thread אחד לכל היותר על המעבד.אם thread ירוץ ללא תכלית על המעבד, הוא ימנע מ- threads אחרים שלהם הוא מחכה מלסיים את פעולתם (עד שיעשו לו context switch). רבים סימנו תשובה נכונה אולם לא ידעו להסביר ועל כך ירדו נקודות.