# Hacettepe University
# Department of Computer Science & Engineering

## BiL236 Programming Laboratory
## Experiment IV

| | |
|---|---|
| **Subject** | **:** Computer Organization |
| **Submission Date** | **:** 12.04.2012 |
| **Due Date** | **:** 30.04.2012 |
| **Programming Language** | **:** Java (Eclipse 3.7 , JDK 1.7) |
| **Advisors** | **:** Asst. Prof. Dr. Harun ARTUNER, R.A. Oğuzhan GÜÇLÜ |

### BACKGROUND INFORMATION

In computing, *multitasking* is a method where multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a *process switch*. When process switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs. [1]

Processes may occupy a variety of states such as *Running*, *Ready* and *Blocked* in execution time (**Figure 1**).

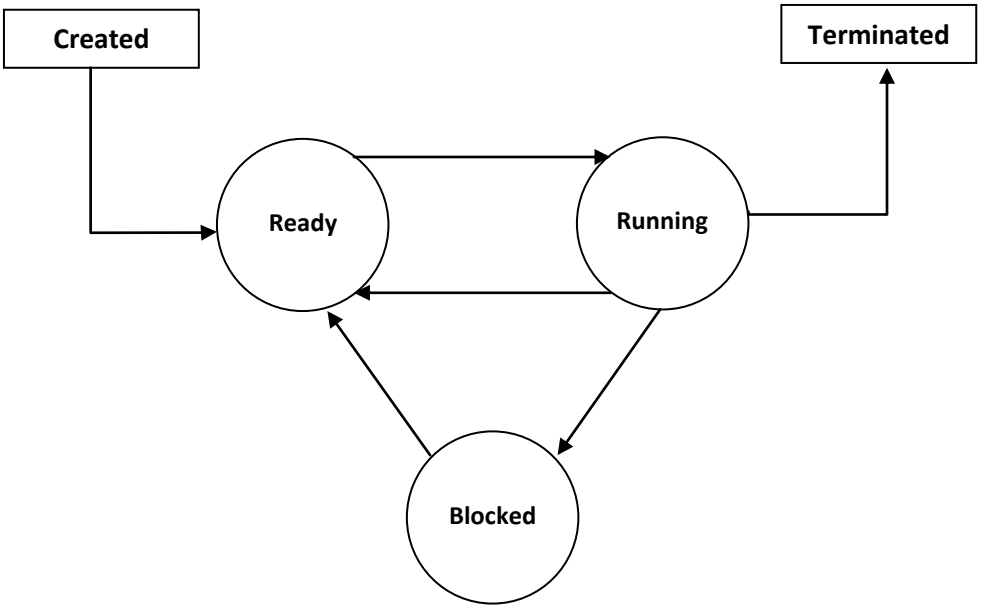


***Figure 1: States of processes***

When a process is created, it awaits admission to the *Ready* state. After entering *Ready* state, the process awaits execution on a CPU (to be switched onto the CPU by a scheduler). There may be many *Ready* processes at any point of the system's execution. For example; in a one-processor system, only one process can be executed at any time, and all other "concurrently executing" processes are waiting for execution in a *Ready Queue*. Other processes waiting for an event to occur, such as *I/O operation* completion (For example; loading information from a hard drive) are in the state of *Blocked* (Thus, they are in the *Blocked Queue*). After the operation is completed, the *Blocked* process leaves *Blocked* state and enters to *Ready*. A process moves into the *Running* state when it is chosen for execution according to process management algorithm. The process' instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU (or core). A running process can leave this state because of some reasons such as starting an *I/O operation*, executing a *suspend* or *wait* command, etc. After leaving *Running* state, the process enters *Ready* or *Blocked* state, according to operation will be carried out. A process may be terminated, from the *Running* state by completing its execution or because of an error condition.

## EXPERIMENT

In this experiment, you are going to develop a simulator for a variety of *IJVM* instructions [2], by handling all operations on macro level. The simulator is going to be run with **one** CPU. The table below contains the instructions you are going to implement.

| Mnemonic | Operands | Description |
|---|---|---|
| BIPUSH | byte | Push a byte onto stack |
| DUP | - | Copy top word on stack and push onto stack |
| ERR | - | Print an error message and terminate |
| GOTO | label_name | Unconditional jump to specified label |
| HALT | - | Print an error message and terminate |
| IADD | - | Pop two words from stack; push their sum |
| IAND | - | Pop two words from stack; push Boolean AND |

| | | |
|---|---|---|
| IFEQ | label_name | Pop word from stack and branch if it is zero |
| IFLT | label_name | Pop word from stack and branch if it is less than zero |
| IF_ICMPEQ | label_name | Pop two words from stack and branch if they are equal |
| IINC | variable_name byte | Add a value to a local variable (First operand is a variable and second is a value, space character(s) between them) |
| ILOAD | variable_name | Push local variable onto stack |
| IN | - | Read an integer from the keyboard buffer and push it onto the stack. If no integer is available, push 0 (zero) |
| IOR | - | Pop two words from stack; push Boolean OR |
| ISTORE | variable_name | Pop word from stack and store in local variable |
| ISUB | - | Pop two words from stack; substract the first top word from the second top word, push the result |
| LDC_W | constant_name | Push constant from constant pool onto stack |
| NOP | - | Do nothing |
| OUT | - | Pop word from stack and print |
| POP | - | Delete word from top of stack |
| SWAP | - | Swap the two top words on the stack |
| SUSPEND | - | Pause execution (the process is going to be moved into *Ready Queue* and another process is going to be switched to CPU) |

***Table 1: Instruction list***

**EXECUTION**

Execution command of the program is going to be in the form below;

**\> java Main.java <*stacksize*> <*memorysize*> <*input-file1*> <*input-file2*> … <*input-fileN*> <*outputfile*>**

| | |
|---|---|
| ***<stacksize>*** | ***:*** Size of allocated stack spaces for each process (for stack usage) |
| ***<memorysize>*** | ***:*** Size of allocated memory spaces for each process (for holding variables and constants) |
| ***<input-fileX>*** | ***:*** Input file containing informations of process *X* |
| ***<output-file>*** | ***:*** Results of process executions are going to be written to output file |

**INPUT FILE FORMAT**

Input file has three sections corresponding *constants*, *variables* and *instructions*. Here is an example;

```
.constant
a 15
b 6
.end-constant

.var
c
d
.end-var

.code

        BIPUSH -5
        LDC_W a
        IADD
        DUP
        DUP
        IFLT Exit
        OUT
        ISTORE c
        BIPUSH 4
        ISTORE d
        ILOAD d
        ILOAD c
        ISUB
Exit:   OUT

.end-code
```

***Figure 2: Input File example***

**OUTPUT FILE FORMAT**

If the program runs with this simple input file using the command below;

**\> java Main.java 10 10 input1.txt output.txt**

Related output file is going to be as follows;

```
PROCESS 1:10
PROCESS 1:-6
PROCESS 1:Finished
All processes have finished
```

***Figure 3: Output File example***

As seen in the example, while printing results, the process has produced that result is specified before. After all instructions are executed, "*PROCESS X:Finished*" message is printed. Finally, when all processes are terminated (*Ready Queue* is empty and there is no *Running* process), "All processes has finished" message is written to output file and simulator stops running.

The example execution above, the program is executed with one process. But what if it runs with more processes? The figure below clarifies this situation.

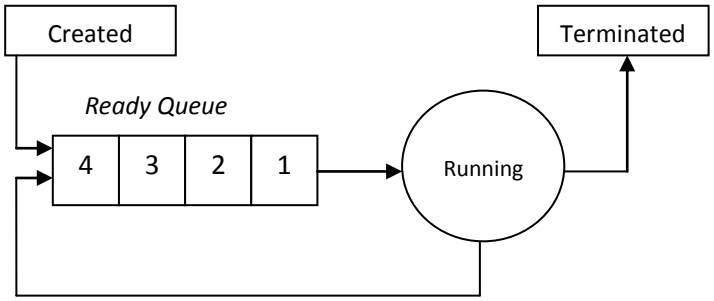


***Figure 4: Process states for example run***

In execution of the program, **you are not going to implement *Blocked State***. All processes are going to be added to Ready Queue, corresponding to their order in the execution command. While executing instructions of a process in the CPU, if a **SUSPEND** instruction comes, the *Running* process is going to be *suspended* and moved to **end of** the *Ready Queue*. Then the first process in the *Ready Queue* is going to leave the queue and move into *Running State*. It is going to be switched to the CPU and start (if this is first switch of that process) or continue (if it is a suspended process) execution. This is the process management algorithm of ***First Come First Served***. If all instructions of a *Running* process are executed or an error occured during execution, the process is going to be terminated (Also related result message is going to be written to output file) and the first process in the *Ready Queue* is going to be switched to the CPU. An example execution like this;

**\> java Main.java 100 100 p1.txt p2.txt p3.txt p4.txt output.txt**

is handled below;

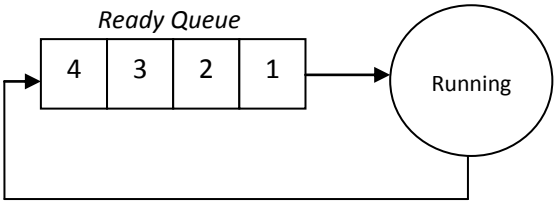


***Figure 5: Four processes are created and added to Ready Queue***

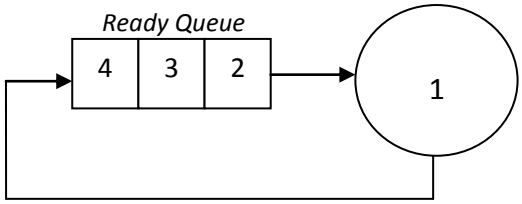


***Figure 6: Process 1 is switched to CPU***

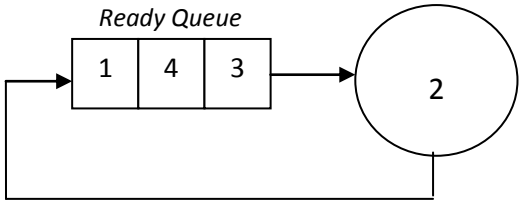


***Figure 7: Process 1 is suspended and Process 2 is switched to CPU***

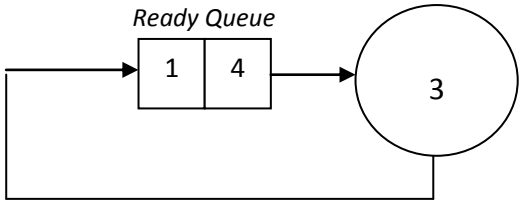


***Figure 8: Process 2 is terminated***

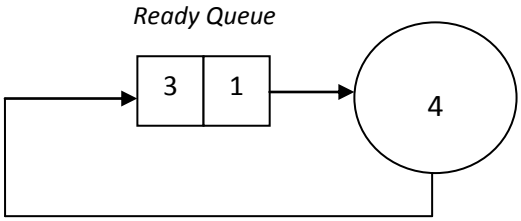


***Figure 9: Process 3 is suspended***

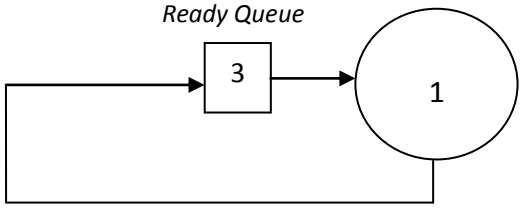


***Figure 10: Process 4 is terminated***

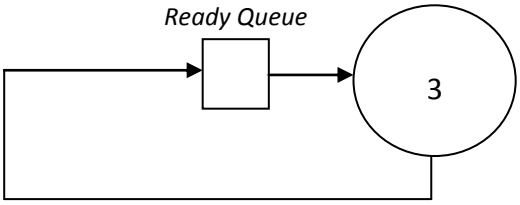


***Figure 11: Process 1 is terminated (No ready process)***

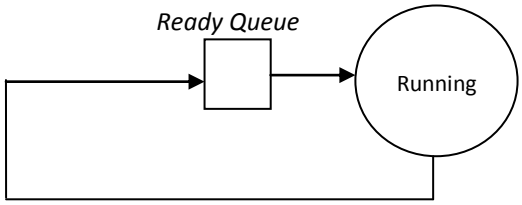


***Figure 12: Process 3 is terminated (No ready or running process) All processes have finished execution***

Also here is a two-process execution example;

```
.constant
a 15
b 6
.end-constant

.var
c
d
.end-var

.code

BIPUSH -5
LDC_W a
IADD
DUP
SUSPEND
DUP
IFLT Exit
OUT
ISTORE c
BIPUSH 4
ISTORE d
SUSPEND
ILOAD d
ILOAD c
ISUB
Exit: OUT

.end-code
```

*Figure 13: input1.txt*

```
.constant
cons 0
.end-constant

.var
A
B
.end-var

.code

BIPUSH 5
ISTORE A
BIPUSH 0
ISTORE B
ILOAD A
ILOAD B
IOR
OUT
SUSPEND
LDC_W cons
POP
GOTO e1
e1: GOTO e2
e2: GOTO e3
e3: BIPUSH 5
BIPUSH 4
OUT
SUSPEND
BIPUSH -5
OUT

.end-code
```

*Figure 14: input2.txt*

If the program runs with these input files using command below;

**\> java Main.java 10 10 input1.txt input2.txt output.txt**

The output file is going to be as follows;

```
PROCESS 2:5
PROCESS 1:10
PROCESS 2:4
PROCESS 1:-6
PROCESS 1:Finished
PROCESS 2:-5
PROCESS 2:Finished
All processes have finished
```

*Figure 15: output.txt*

There are some other important points;

- Byte and word operands are integer values
- **All exception types must be handled in your implementation.** For example, stack overflow or underflow, memory overflow, syntax error, wrong operand number or type, etc. You can detect any exception before or during execution, the important

point is that, an error message related to the exception must be written to output file, and if exception occurs during execution, that process must be terminated.

- The Simulator must run in **case-insensitive** mode. For example, the variables has name “abc” and “ABC” are the **same**.
- All print operations are going to be made on the **output file**. (For instructions which are printing messages --*ERR*, *HALT* and *OUT*-- and error messages). **No message will be printed on screen.**
- The **<memorysize>** parameter is the maximum number of constants and variables (sum of them) can be hold on memory for each process.
- Remember that the Simulator is going to run with **one** CPU. **You are not supposed to use threads for implementation.**

## NOTES

- You will use online submission system to submit your experiments. https://submit.cs.hacettepe.edu.tr/ (No other submission method such as diskette, CD or email will be accepted)
- Your submission code file structure is going to be announced later.
- Submission time for deadline is: 17:00.
- Do not submit any file via e-mail related with this assignment.
- **SAVE** all your work until the assignment is graded.
- The assignment must be original, **INDIVIDUAL** work. Duplicate or very similar assignments are both going to be punished. General discussion of the problem is allowed, but **DO NOT SHARE** your design.
- You can ask your questions through course’s news group: news://news.cs.hacettepe.edu.tr/dersler.bil236 And you are supposed to be aware of everything discussed in the newsgroup.

## REFERENCES


**1.** http://en.wikipedia.org/wiki/Computer_multitasking

**2.** http://en.wikipedia.org/wiki/IJVM

**3.** Bilgisayar İşletim Sistemleri, *Prof. Dr. Ali SAATÇİ*

**4.** Structured Computer Organization, *Andrew S. Tannenbaum*

**5.** http://www.eclipse.org/downloads/packages/eclipse-classic-372/indigosr2

**6.** http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u3-download-1501626.html