Tutorial I-a:

1. What are the three main purposes of an operating system?

Answer:

- To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
- To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as fair and efficient as possible.
- As a control program it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and control of I/O devices.

2. List the four steps that are necessary to run a program on a completely dedicated machine.

Answer:

i. Reserve machine time.
ii. Manually load program into memory.

iii. Load starting address and begin execution.
iv. Monitor and control execution of program from console.

3. Define the essential properties of the following types of operating systems:
a. Batch
b. Interactive
c. Time sharing
g. Distributed

Answer:
a. Batch. Jobs with similar needs are batched together and run through
the computer as a group by an operator or automatic job sequencer.
Performance is increased by attempting to keep CPU and I/O devices
busy at all times through buffering, off-line operation, spooling,
and multiprogramming. Batch is good for executing large jobs
that need little interaction; they can be submitted and picked up
later.

b. Interactive. This system is composed of many short transactions
where the results of the next transaction may be unpredictable.
Response time needs to be short (seconds) since the user submits
and waits for the result.

c. Time sharing. This system uses CPU scheduling and multiprogramming
to provide economical interactive use of a system. The
CPU switches rapidly from one user to another. Instead of having
a job defined by spooled card images, each program reads its next
control card from the terminal,
g. Distributed.This system distributes computation among several
physical processors. The processors do not share memory or a
clock. Instead, each processor has its own local memory. They communicate
with each other through various communication lines,

such as a high-speed bus or local area network.

4. What are the main differences between operating systems for mainframe computers and personal computers?

Answer:
Generally, operating systems for batch systems have simpler requirements than for personal computers. Batch systems do not have to be concerned with interacting with a user as much as a personal computer. As a result, an operating system for a PC must be concerned with response time for an interactive user. Batch systems do not have such requirements. A pure batch system also may have not to handle time sharing, whereas an operating system must switch rapidly between different jobs.

5. How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) system?

Answer:
The distinction between kernel mode and user mode provides a rudimentary form of protection in the following manner. Certain instructions could be executed only when the CPU is in kernel mode. Similarly, hardware devices could be accessed only when the program is executing in kernel mode. Control over when interrupts could be enabled or disabled is also possible only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode, thereby enforcing protection of critical resources.

6. What is the purpose of interrupts? What are the differences between a
trap and an interrupt? Can traps be generated intentionally by a user
program? If so, for what purpose?
Answer: An interrupt is a hardware-generated change of flow within
the system. An interrupt handler is summoned to deal with the cause
of the interrupt; control is then returned to the

interrupted context and
instruction. A trap is a software-generated interrupt. An interrupt can
be used to signal the completion of an I/O to obviate the need for device
polling. A trap can be used to call operating system routines or to catch
arithmetic errors.

Tutorial I-b:

1. Which of the following instructions should be privileged?

a. Set value of timer. (p)
b. Read the clock.
c. Clear memory. (p)
d. Turn off interrupts. (p)
e. Switch from user to kernel mode.

2. Give two reasons why caches are useful.What problems do they solve?

What problems do they cause? If a cache can be made as large as the
device for which it is caching (for instance, a cache as large as a disk),
why not make it that large and eliminate the device?
Answer: Caches are useful when two or more components need to exchange
data, and the components perform transfers at differing speeds.
Caches solve the transfer problem by providing a buffer of intermediate
speed between the components. If the fast device finds the data it needs
in the cache, it need not wait for the slower device. The data in the cache
must be kept consistent with the data in the components. If a component
has a data value change, and the datum is also in the cache, the cache
must also be updated. This is especially a problem on multiprocessor
systems where more than one process may be accessing a datum. A component
may be eliminated by an equal-sized cache,

but only if: (a) the
cache and the component have equivalent state-saving capacity (that is,
if the component retains its data when electricity is removed, the cache
must retain data as well), and (b) the cache is affordable, because faster
storage tends to be more expensive.

3. Writing an operating system that can operate without interference from (malicious or undebugged) user programs requires some hardware assistance. Name three hardware aids for writing an operating system, and describe how they could be used together to protect the operating system.

Answer:

a. Monitor/user mode b. Privileged instructions c. Timer d. Memory protection

4. What are the major activities of an operating system in regard to the following components?

(1) Process management. (2) Memory management. (3) Secondary-storage management.
(4) File management.

Answer: text p67 (8th ed), 74 (9th ed)

(a) What are the five major activities of an operating system in regard to process management?

1. The creation and deletion of both user and system processes
2. The suspension and resumption of processes
3. The provision of mechanisms for process synchronization
4. The provision of mechanisms for process communication
5. The provision of mechanisms for deadlock handling

(b) What are the three major activities of an operating system in regard to memory management?

Answer:

i. Keep track of which parts of memory are currently being used and by whom.
ii. Decide which processes are to be loaded into memory when memory space becomes available.
iii. Allocate and deallocate memory space as needed.

(c) What are the three major activities of an operating system in regard to secondary-storage management?

Answer:
i. Free-space management.
ii. Storage allocation.
iii. Disk scheduling.

5. What are the purposes of system calls and programs?
Answer:
System calls allow user-level processes to request services of the operating system. . System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need

to write their own programs to solve common problems.

System call is a software interrupt that changes the processor from user to monitor/kernel mode, so that a user program can obtain system services through privileged instructions executed on its behalf *by the operating system*. System calls, for example, are usually needed for a user program to perform various kinds of IO operations.

6. What are the main advantages of using the following approaches to system design?
(1) Layered approach (2) Separation of mechanism and policy
(3) Virtual-machine architecture

Answer:
(1) As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes

affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

(2) Mechanism and policy must be separate to ensure that systems
are easy to modify. No two system installations are the same, so each installation may want to tune the operating system to suit its needs.
With mechanism and policy separate, the policy may be changed at will
while the mechanism stays unchanged. This arrangement provides a
more flexible system.

(3) The system is easy to debug, and security problems are easy
to solve. Virtual machines also provide a good platform for operating
system research since many different

operating systems can run on one
physical system.

Tutorial II-a:

Q.1
Answer: • A method of time sharing must be implemented to allow each of several processes to have
access to the system. This method involves the preemption of processes that do not voluntarily give up the CPU (by using a system call, for instance) and the kernel being reentrant (so more than one process may be executing kernel code concurrently).
• Processes and system resources must have protections and must be protected from each other.
Any given process must be limited in the amount of memory it can use and the operations it
can perform on devices like disks.
• Care must be taken in the kernel to prevent

deadlocks between processes, so processes aren’t
waiting for each other’s allocated resources.

Q.2
Answer: Threads are very inexpensive to create and destroy, and they use very little resources while they
exist. They do use CPU time for instance, but they don’t have totally separate memory spaces.
Unfortunately, threads must ‘‘trust’’ each other to not damage shared data. For instance, one
thread could destroy data that all the other threads rely on, while the same could not happen
between processes unless they used a system feature to allow them to share data. Any program
that may do more than one task at once could benefit from multitasking. For instance, a program
that reads input, processes it, and outputs it could have three threads, one for each task.

‘‘Singleminded’’
processes would not benefit from multiple threads; for instance, a program that displays the time of day.

Q.3
Answer: User-level threads have no kernel support, so they are very inexpensive to create, destroy, and
switch among. However, if one blocks, the whole process blocks. Kernel-supported threads are
more expensive because system calls are needed to create and destroy them and the kernel must
schedule them. They are more powerful because they are independently scheduled and block
individually.

Q.4
Answer: The shared data structures are as in the solution presented in Section 4.6, with the addition of:
var *full*: **array**[0..n-1] **of** *boolean*

initially *full*[*i*] = *false*, for all *i*.
The producer process has a local variable *nextp* in which the new item to be produced is stored:

repeat
...
produce an item in *nextp*
...
while *full*[*in*] **do** *skip*;
buffer[*in*] := *nextp*;
full[*in*] := *true*;
in := *in*+1 **mod** *n*;
until *false*;

The code for the consumer process can be modified as follows:

repeat
while not *full*[*out*] **do** *skip*;
nextc := *buffer*[*out*];
full[*out*]:= *false*;
out := *out*+1 **mod** *n*;
...
consume the item in *nextc*
...
until *false*;

Q.5
Answer: Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the
CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process
relinquishes control of the CPU only when it finishes with its current CPU burst.

Q.6
Answer: Exercises Solution 5.12 (text p.220)
====

Tutorial II-b

Q.1.
Answer: Practice Solution 5.3 (text p. 219)

Q.2
Answer: Exercises Solution 5.14 (text p.221)

Q.3

Answer: Practice Solution 5.4 (text p. 219)

Q.4
Answer: Practice Solution 5.5 (text p. 219)
===

Tutorial II-c

Q.1
Answer: Suppose the value of semaphore *S* = 1 and processes P1 and P2 execute *wait*(*S*) concurrently.
1. T0: P1 determines that value of *S* = 1
2. T1: P2 determines that value of *S* = 1
3. T2: P1 decrements *S* by 1 and enters critical section
4. T3: P3 decrements *S* by 1 and enters critical section

Q.2 (Practice 6.2)
Answer: The shared data structures are
var *a*: array [0...2] *of* semaphore {initially = 0}

agent: semaphore {initially = 1}
The agent process code is as follows:
repeat
Set *i*, *j* to a value between 0 and 2.
wait(*agent*);
signal(*a*[*i*]);
signal(*a*[*j*]);
until *false*;
Each smoker process needs two ingredients represented by integers *r* and *s* each with value between 0 and 2.
repeat
wait(*a*[*r*]);
wait(*a*[*s*]);
‘‘smoke’’
signal(*agent*);
until *false*;

Q.3
Answer: The following construct that transforms the *wait* and *signal* operations on a semaphore *S* into
equivalent critical regions, proves that semaphores are as powerful as critical regions.
A

semaphore *S* is represented as a shared integer;

var *S*: **shared** integer;

The *wait*(*S*) and *signal*(*S*) operations are implemented as follows:

wait(*S*): **region** *S* **when** *S* > 0 **do** *S* := *S* - 1;

signal(*S*): **region** *S* **do** *S* := *S* + 1;

The implementation of critical regions in terms of semaphores in Section 6.6 proves that critical regions are as powerful as semaphores. The implementation of semaphores in terms of monitors in Section 6.7 proves that semaphores are as powerful as monitors. The proof that
monitors are as powerful as semaphores follows from the following construct:

type *semaphore* = **monitor**
var *busy*: *boolean*;
nonbusy: *condition*;
procedure entry *wait*;
begin
if *busy* **then** *nonbusy.wait*;
busy := *true*;
end;
procedure entry *signal*;

begin
busy := *false*;
nonbusy.signal;
end;
begin
busy := *false*;
end.

Q.4 (Exercises 6.28)
Answer: The monitor code is
type *resource* = **monitor**
var *P*: **array**[0..2] **of** *boolean*;
X: *condition*;
procedure *acquire* (*id: integer*, *printer-id: integer*);
begin
if *P* [0] **and** *P* [1] **and** *P* [2] **then** *X.wait*(*id*)
if not P[0] **then** *printer-id* := 0;
else if not P[1] **then** *printer-id* := 1;
else *printer-id* := 2;
P [*printer-id*] := *true*;
end
procedure *release* (*printer-id: integer*)
begin
P [*printer-id*] := *false*;

X.signal;
end
begin
P [0] := *P* [1] := *P* [2] := *false*;
end
===

Tutorial II-d

Q.1 (exercises 7.10)
Answer: a. Each section of the street is considered a resource.
• **Mutual-Exclusion**— only one vehicle on a section of the street.
• **Hold-and-wait** — each vehicle is occupying a section of the street and is waiting to move to the next section.
• **No-preemption** — a section of a street that is occupied by a vehicle cannot be taken away from the vehicle unless the car moves to the next section.
• **Circular-wait** — each vehicle is waiting for the next vehicle in front of it to move.

b. Allow a vehicle to cross an intersection only if it is assured that the vehicle will not have to
stop at the intersection.

Q.2 (exercises 7.15 & 7.16)
Answer: see solution manual.

Q.3 (exercises 7.20)
Answer: see solution manual.

Q.4 (practice ex 7.8)
Answer: see solution manual.
===

Tutorial III-a

Q.1(exercises 8.11)
Answer: see solution manual.

Q.2(exercises 8.20)
Answer: see solution manual.

Q.3(practice ex 8.5)
Answer: see solution manual.

Q.4(exercises 8.21)
Answer: see solution manual.

Q.5(exercises 8.23)
Answer: see solution manual.

Q.6(practice ex 8.8)
Answer: see solution manual.
===

Tutorial III-b

Q.1(practice ex 9.3)
Answer: see solution manual.

Q.2(exercises 9.21)
Answer: see solution manual.

Q.3(practice ex 9.8)
Answer: see solution manual.

Q.4(practice ex 9.11)
Answer: see solution manual.

Q.5(practice ex 9.13)
Answer: see solution manual.

Q.6(exercises 9.33)
Answer: see solution manual.
===

Tutorial IV-a

Q.1
Answer: If arbitrarily long names can be used then it is possible to simulate a multilevel directory structure.
This can be done, for example, by using the character ‘‘.’’ to indicate the end of a subdirectory.
Thus, for example, the name *jim.pascal.F1* specifies that *F1* is a file in subdirectory *pascal* which

in turn is in the root directory *jim*.
If file names were limited to seven characters, then the above scheme could not be utilized and thus, in general, the answer is *no*. The next best approach in this situation would be to use a
specific file as a symbol table (directory) to map arbitrarily long names (such as *jim.pascal.F1*)
into shorter arbitrary names (such as *XX00743*), which are then used for actual file access.

Q.2
Answer: With a single copy, several concurrent updates to a file may result in user obtaining incorrect
information, and the file being left in an incorrect state. With multiple copies, there is storage
waste and the various copies may not be consistent with respect to each other.

Q.3
Answer: a. There are two methods for

achieving this:
i. Create an access control list with the names of all 4990 users.
ii. Put these 4990 users in one group and set the group access accordingly. This scheme cannot always be implemented since user groups are restricted by the system.
b. The universe access information applies to all users unless their name appears in the accesscontrol
list with different access permission. With this scheme you simply put the names of
the remaining ten users in the access control but with no access privileges allowed.
Q.4(practice ex 11.4)
Answer: see solution manual.

Q.5(exercises 11.14)
Answer: see solution manual.
===

Tutorial IV-b

Q.1(exercises 12.15)
Answer: see solution manual.

Q.2
Answer: a. FCFS -- 565
(143 -> 86 -> 147 -> 91 -> 177 -> 94 -> 150 -> 102 -> 175 -> 130)
b. SSTF -- 162
(143 -> 147 -> 150 -> 130 -> 102 -> 94 -> 91 -> 86 -> 175 -> 177)
c. SCAN -- 169
(143 -> 147 -> 150 -> 175 -> 177 -> 199 -> 130 -> 102 -> 94 -> 91 -> 86)
d. LOOK -- 125
(143 -> 147 -> 150 -> 175 -> 177 -> 130 -> 102 -> 94 -> 91 -> 86)
e. C-SCAN -- 186
(143 -> 147 -> 150 -> 175 -> 177 -> 199 -> 0 -> 86 -> 91 -> 94 -> 102 -> 130)

Q.3
Answer: We provide only the solution for C-SCAN.
type *diskhead* = **monitor**
var *busy*: *boolean*;

```
up: condition;
headpos, count: integer;
procedure entry acquire (dest: integer);
begin
if busy
then if headpos £ dest
then up.wait(dest + count)
else up.wait(dest + count+n);
busy := true;
if dest < headpos
then count := count + n;
headpos := dest;
end;
procedure entry release;
begin
busy := false;
up.signal;
end;
begin
headpos := 0;
count := 0;
end.
```

Q.4(practice ex 12.2)
Answer: see solution manual.

Q.5(exercises 12.22 a & b)
Answer: see solution manual.

Q.6(practice ex 12.6)
Answer: see solution manual.
===

Tutorial V

Q.1(practice ex 14.3)
Answer: see solution manual.

Q.2(exercises 14.13)
Answer: see solution manual.

Q.3(practice ex 14.6)
Answer: see solution manual.

Q.4(exercises 15.4)
Answer: see solution manual.

Q.5(exercises 15.7)

Answer: see solution manual.

Q.6(exercises 15.10)
Answer: see solution manual.