1. Suppose that you wish to design a virtual memory system with the following characteristics:

   • The size of a page table entry is 4 bytes.
   • Each page table must fit into a single physical frame.
   • The system must be able to support virtual address spaces as large as $2^{38}$ bytes (256 GB).

Suppose that you decide to use a multi-level paging scheme with no more than two levels of page tables. What is the minimum page size that your system must have?

**Answer:**

Page table entry 4 bytes = $2^2$ bytes
Virtual address space = $2^{38}$ bytes
Page size, $2^p$ = ?

Here,

1- level page needs 1 physical frame
2- level page needs 1 physical frame
Multi- level page needs 1 physical frame

As we know, for two-level page table,

$$(2^p-2) (2^p-2) (2^p) = 2^{38}$$

Hence, $(P-2) + (P-2) + P = 38$

Or, $3P - 4 = 38$

Or, $3P = 38 + 4 = 42$

Or, $P = 14$

So, Page size = $2^{14}$ bytes
2. Measurements of a certain system have shown that the average process runs for a time $T$ before blocking on I/O. A process switch requires a time $S$, which is effectively wasted (overhead). For round-robin scheduling with quantum $Q$, give a formula for the CPU efficiency for each of the following:

(a) $Q = \infty$
(b) $Q \geq T$
(c) $S < Q < T$
(d) $Q = S$
(e) $Q$ nearly 0

Answer:

Here, $T =$ Average Process Running Time
$S =$ Process Switching time
And $Q =$ Quantum

The CPU efficiency is the useful CPU time divided by the total CPU time. When $Q \geq T$, the basic cycle is for the process to run for $T$ and undergo a process switch for $S$.

The CPU efficiency here is then

*a) $Q = \infty$

CPU efficiency = Efficient running time / Total Time
= $T / (T+S)$

*b) $Q \geq T$

Hence,

CPU efficiency = Efficient running time / Total Time
= $T / (T+S)$

*c) $S < Q < T$

When the quantum is shorter than $T$, each run of $T$ will require $T/Q$ process switches, wasting a time is $ST/Q$.

Hence,

CPU efficiency = Efficient running time / (Total Time + Total wasting time)
= $T / (T+ST/Q)$
= $T / T(1 + S/Q)$
= $Q / (Q+S)$

*d) $Q=S$

We just substitute $Q$ for $S$ and find that the efficiency is 50%.
3. Suppose that an operating system supports two kinds of sequential processes: high-priority interactive processes, and low-priority non-interactive processes. The behavior of the high-priority processes is to alternate between periods of computation of duration \(t_c\) and periods of blocking (waiting for input) of duration \(t_b\). The behavior of the low-priority processes is to compute constantly, with no blocking. The operating system’s scheduling policy is prioritized round-robin with a quantum \(q\), where \(t_b < q\). Scheduling decisions are made only when a quantum expires, or when the running process blocks. The scheduler selects a low-priority process to run only if no high-priority processes are ready. Suppose there is one high-priority process and one low-priority process in the system, and that both processes will run for a long time. For what fraction of the time does the low-priority process run?

**Answer:**

Once the high priority process blocks (after running for time \(t_c\)), the low priority process will run for a quantum. If the high priority process is still blocked after that quantum, the low priority process will receive another.

When the high priority process unblocks, the low priority process will finish out the current quantum, at which point the scheduler will give the processor to the high priority process.

Thus, for every \(t_c\) time the high priority process runs the low priority process and will run for \([t_b/q]q\) units of time. The fraction of CPU time used by the low priority process is

\[
\frac{[t_b/q]q}{[t_b/q]q + t_c}
\]

4. Can a process make a transition from the Ready state to the Blocked state? Why or why not?

**Answer:**

No, cannot. A process can become blocked, only then it issues a request for a resource that is not available and it can make such a request only when it is executing.

5. Explain the distinction between a real address and a virtual address.
**Answer:**

A virtual address refers to a memory location in virtual memory. That location is on disk and at some times in main memory. A real address is an address in main memory.

6. Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each of which needs 10 minutes of CPU time, start simultaneously. Assume 50% I/O wait.

   a) How long will the last one take to complete if they run sequentially?
   b) How long if they run in Parallel?

**Answer:**

If each job has 50% I/O wait, then it will take 20 minutes to complete in the absence of competition.

If run sequentially, the second one will finish 40 minutes after the first one starts.

With two jobs, the approximate CPU utilization is \((1 - 0.5^2)\) or 0.75. Thus each one gets 0.375 CPU utilization per minute of real time.

To accumulate 10 minutes of CPU time, a job must run for \((10/0.375)\) or about 26.67 minutes.

Thus,

a) Running sequentially the jobs finish after 40 minutes.

b) Running in parallel they finish after 26.67 minutes.

7. Explain the difference between a monolithic kernel a microkernel and a real-time kernel.

**A. Monolithic Kernel**

A monolithic kernel is a large kernel containing virtually the complete operating system, including scheduling, file system, device drivers, and memory management. All the functional components of the kernel have access to all of its internal data structures and routines. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space.
Monolithic kernels have a distinction between the user and kernel space. When software runs in the user space normally it cannot access the system hardware nor can it execute privileged instructions. Using special entry points (provided by hardware), an application can enter the kernel mode from user space. The user space programs operate on a virtual address so that they cannot corrupt another application’s or the kernel’s memory. However, the kernel components share the same address space; so a badly written driver or module can cause the system to crash. Figure A, shows the architecture of monolithic kernel where the kernel and kernel sub modules share the same address space and where the applications each have their private address spaces.

B. Microkernel

A microkernel is a small privileged operating system core that provides process scheduling, memory management, and communication services and relies on other processes to perform some of the functions traditionally associated with the operating system kernel.

These kernels have been subjected to lots of research especially in the late 1980s and were considered to be the most superior with respect to OS design principles. However, translating the theory into practice caused too many bottlenecks; very few of these kernels have been successful in
the marketplace. The microkernel makes use of a small OS that provides the very basic service (scheduling, interrupt handling, message passing) and the rest of the kernel (file system, device drivers, networking stack) runs as applications. On the usage of MMU, the real-time kernels form one extreme with no usage of MMU whereas the microkernels are placed on the other end by providing kernel subsystems with individual address space. The key to the microkernel is to come up with well-defined APIs for communication with the OS as well as robust message-passing schemes. Figure B shows a microkernel architecture where kernel subsystems such as network stack and file systems have private address space similar to applications. Microkernels have been vigorously debated especially against the monolithic kernels. One such widely known debate was between the creator of Linux, Linus Torvalds, and Andrew Tanenbaum who was the creator of the Minix OS (a microkernel).

C. Real-Time Kernel

Traditional real-time kernels are meant for MMU-less processors. On these operating systems, the entire address space is flat or linear with no memory protection between the kernel and applications, as shown in Figure C. Figure C shows the architecture of the real-time executive where the core kernel, kernel subsystems, and applications share the same address space.

These operating systems have small memory and size footprint as both the OS and applications are bundled into a single image. As the name suggests, they are real-time in nature because there is no overhead of system calls, message passing, or copying of data. Adding new software becomes a not-so-pleasant action because it needs to be tested thoroughly that it brings down the entire system. Also it is very difficult to add applications or kernel modules dynamically as the system has to be brought down. Most of the proprietary and commercial RTOSs fall under this category.


8. What is the purpose of system calls, and how do system calls relate to the OS and to the concept of dual-mode (kernel mode and user mode) operation?
A system call is used by an application program to invoke a function provided by the operating system. Typically, the system call results in transfer to a system program that runs in kernel mode.