# **BCIT**

**Comp 4735 Operating Systems**

**Instructor: Mirela Gutica**

**Fall 2015**

Mark: \_\_\_\_\_\_\_\_ 100

Assignment 1

Note: To receive any credit whatsoever, your answers must be legible and readily readable in the judgement of the grader. Add brief explanatory comments as necessary to make sure your answers are clear and unambiguous to the grader. **When you solve a problem, show all the steps, similar with the examples in the lectures**. Just the answer will not give you credits for a problem. **It is required to have a professional layout for the assignment**.

The assignment should be handed-in (on D2L) no later than **11:30pm,** **Tuesday, October 13, 2015. No late assignments will be accepted.**

If not specified, a question weights **10 marks**.

**Answer to questions:**

(20p)

1. Consider the Intel i7 processor (64 and IA-32 architectures). Use latest documentation for this processor to answer the following questions:
   1. What data types are supported by the Intel processors? Enumerate.

The fundamental data types that are supported by the Intel processors are as follows:

* + 1. Byte
    2. Word
    3. Doubleword
    4. Quadword
    5. Double Quadword

The numeric data types are as follows:

1. Byte Unsigned Integer
2. Word Unsigned Integer
3. Doubleword Unsigned Integer
4. Quadword Unsigned Integer
5. Byte Signed Integer
6. Word Signed Integer
7. Doubleword Signed Integer
8. Quadword Signed Integer
9. Half Precision Floating Point
10. Single Precision Floating Point
11. Double Precision Floating Point
12. Double Extended Precision Floating Point

The pointer data types are as follows:

1. Near pointer
2. Far pointer

Bit field data types are as follows:

1. Bit Field

String data types are as follows:

1. Bit string
2. Byte string

Packed SIMD data types are as follows:

1. 64-bit
   1. Packed Bytes
   2. Packed Words
   3. Packed Doublewords
   4. Packed Byte Integers
   5. Packed Word Integers
   6. Packed Doubleword Integers
2. 128-bit
   1. Packed bytes
   2. Packed words
   3. Packed Doublewords
   4. Packed Quadwords
   5. Packed Single Precision Floating Point
   6. Packed Double Precision Floating Point
   7. Packed Byte Integers
   8. Packed Word Integers
   9. Packed Doubleword Integers
   10. Packed Quadword Integers

BCD and Packed BCD Integers are as follows:

1. BCD Integers
2. Packed BCD Integers
3. 80-bit Packed BCD Decimal Integers
   1. State and describe the stack manipulation instructions. State the role of each instruction.

The PUSH, POP, PUSHA, and POPA instructions transfer data onto and off of the process’s stack.

**PUSH**

The PUSH instruction goes into the ESP (stack pointer) register and decrements the value contained inside of it. It operates on memory operands, immediate operands, and register operands, including segment registers. This instruction is typically called to place parameters on the stack before calling a procedure, or to be used to reserve space on the stack for temporary variables.

**POP**

The POP instruction copies the contents of the top of the stack to the location specified with the destination operand, and increments the ESP register to point to the new top of stack. The destination operand may specify a general-purpose register, a segment register, or a memory location.

**PUSHA**

Allows for only one instruction to save all of 8 general-purpose registers on the stack, as opposed to having to issue one save instruction for each register. The order of registers pushed onto the stack are as follows:

1. EAX
2. ECX
3. EDX
4. EBX
5. The initial value of ESP before EAX was pushed
6. EBP
7. ESI
8. EDI

**POPA**

The POPA instruction is essentially the reverse instruction of PUSHA. It copies the top eight words into the general-purpose registers, EXCEPT the ESP register. The order of words on the stack that are copied to the registers are as follows:

* + 1. DI
    2. SI
    3. BP
    4. Ignore word
    5. BX
    6. DX
    7. CX
    8. AX
  1. The CALL instruction (used to call a function/procedure) has two flavours: *near call* and *far call*. What is the difference between the two?

A near call is a call to a procedure within the current code segment. A far call is a call to a procedure located in a different segment other than the current code segment.

* 1. Besides the method of passing parameters to the stack, the documentation indicates two other methods. Give a short description for each.

**Through General-Purpose Registers**

Instead of storing the passed parameters on the stack, the parameters can be copied to general-purpose registers instead, and the next procedure can access these values from the registers.

**In an Argument List**

A data structure in the form of some list is created in the memory, and a pointer to that list is placed on either the stack or a general-purpose register.

* 1. Give descriptions for interrupts and exceptions underlying the difference between them.

**Interrupt**

An interrupt is an asynchronous signal from an external event that causes the current program operation to halt.

**Exception**

A synchronous event that is triggered when the processor finds one or more conditions met while executing an instruction.

* 1. What exception classes exist for this processor?
     1. Give a short description for each.
     2. Explain if the OS terminates the process or if the problem is correctable.

**Faults**

These can be corrected and the program may continue as if nothing happened. When a fault exception occurs, certain registers which are pushed on the stack point to the address of the instruction which generated the exception. This gives the exception handler a chance to fix the condition which caused the exception to occur, before restarting the faulting instruction.

**Aborts**

Traps are reported immediately after the execution of the trapping instruction. Aborts neglect to specify the location of the faulting instruction, since they are usually used to indicate severe errors (such as hardware errors or illegal system tables) which are not recoverable.

**Traps**

Traps are similar to interrupts in the sense that they make the processor push the address of the next instruction to the stack

Hint: Find the answers in the technical documentation (e.g., <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf> ).

1. Give all four situations and reasons for a process to be moved to the state Ready/ Suspend. For each case, give an example. Refer to the figure 3.9b from the textbook – Process State Transition Diagram with Suspended States.

**Blocked/Suspended 🡪 Ready/Suspended**

A process on the disk has received the requested event for which it was initially waiting, except that there are processes that have a higher priority which would remain in the memory.

Example: Windows Update initially is running in the background. With automatic updates enabled, it will download and install the updates in the background.

**Ready 🡪 Ready/Suspended**

A process is moved to the disk if the system runs out of main memory, but is still ready for the dispatcher.

Example: System runs out of memory because the user tries to execute too many processes at once.

**New 🡪 Ready/Suspended**

When a new process is created and is ready to be executed, it may be placed on the disk if there is not enough memory for it to reside.

Example: The main memory is already full when the user tries to execute a new process.

**Running 🡪 Ready/Suspended**

When a process has finished its time allocation, it may be moved to the disk to free up some memory for other higher-priority processes.

Example: The user stops using Google Chrome for an extended period of time; as a result, the OS moves the process to the disk, but it is still ready to be accessed by the user.

(20p)

1. Consider an operating system that executes within user processes. Consider (1) a procedure call that passes in two arguments and returns a result, and (2) a system call that passes in one argument of type char and displays the character (e.g., “E”) on the screen. Both the procedure and the system call are executed in process P1, and each take a little less than a time slice. Consider that after P1, process P2 is in the ready queue. After execution, P1 will join the ready queue. No other processes are in the ready queue.

Using diagrams and explanations answer the following questions:

1. Enumerate the steps performed in the case of the procedure call. Assume that the procedure is called during the first half of a time slice and returns in the same time slice.
   * 1. Push the address of next instruction on the stack
     2. Push the two arguments on the stack
     3. Push the address of where the result will go in the memory
     4. Perform the function, creating any temporary variables in the stack
     5. Push the result to the temporary variable
     6. Pop the result off the stack to the specified address
     7. Pop everything off and go to the next instruction
2. Enumerate the steps performed in the case of the procedure call. Assume that the procedure is called during the second half of a time slice and returns in the next time slice assigned to P1.
3. Push the address of next instruction on the stack
4. Push the two arguments on the stack
5. Push the address of where the result will go in the memory
6. Timeout: Dispatcher saves process P1 context (PCB, registers, etc)
7. Dispatcher loads process P2 context (PCB, registers, etc)
8. Dispatcher executes P2 context
9. Timeout: Dispatcher saves process P2 context
10. Dispatcher loads process P1 context
11. Execute P1 context
12. Perform the function, creating any temporary variables in the stack
13. Push the result to the temporary variable
14. Pop the result off the stack to the specified address
15. Pop everything off and go to the next instruction
16. Enumerate the steps performed in the case of the system call. Consider that the system call is called during the second half of a time slice and finishes in the next time slice assigned to P1.
17. Switch to kernel mode
18. Push the address of the next instruction onto the kernel stack
19. Push the char on the kernel stack
20. Timeout: Dispatcher Saves process P1 context (PCB, registers, etc)
21. Dispatcher Loads process P2 context (PCB, registers, etc)
22. Dispatcher Executes P2 context
23. Timeout: Dispatcher Saves process P2 context
24. Dispatcher Loads process P1 context
25. Execute P1 context
26. Execute the OS function to display “E”
27. Switch to User Mode
28. Which is more expensive: the procedure call or the system call if a) return is in the same time slice b) return is in a different time slice? Why? Elaborate.

If the return is in the same time slice, the system call is more expensive because you have to switch to Kernel mode. If the return is in a different time slice, then the system call would be less expensive: Since the dispatcher needs to execute in kernel mode for both procedures, this second scenario would save time from switching back and forth from user mode to kernel mode when making a system call; both the process and the dispatcher are already in kernel mode.

1. Disuses what an operating system should do when a new process is created and when a process terminates.

To create a new process, the OS should first assign a unique process identifier to the new process, and allocate space for the process. It then creates and initializes the process’s process control block, then sets any appropriate linkages, and creates or expands other data structures. When a process terminates, resources are deallocated.

1. In many modern operating systems, when an interrupt or a system call in a process transfers control to the operating system, a kernel stack area separate from the user stack area is created. Why? Give at least two reasons. Elaborate.

It is important for a process’s image to separate the kernel stack area from the user stack is for security reasons. The kernel area has elevated privileges, which gives it access to the OS’s Kernel functions. It is important to not mix up your User Level privileges and Kernel Level Privileges.

Kernel Stacks also optimize space. Sometimes a process may never make system calls, and as such the process will not need to spend time allocating resources for a stack it would never use.

1. Contrast and compare ULT and KLT. What solution is better and in what circumstances?

**User Level Threads**

With user-level threads, all thread management is done by the application; the programmer and application are in charge of managing these threads. The kernel and ULT are separated; the ULT is in user-level space and is controlled with the thread library, and the kernel lives in kernel space, and the kernel is unaware of ULTs.

The kernel treats the process a­­­­­­s a single unit, so when the dispatcher switches processes, the process’s currently active user-level thread will remain in its running state, even if the process is not currently running on the CPU. When the process resumes running on the CPU, so will the thread in the running state.

User-Level Threads are faster than Kernel-Level Threads if there are no system calls being made; the process can switch ULTs much more quickly because it doesn’t have to communicate with the Kernel directly, and instead only communicates with the Thread Library.

One disadvantage of ULTs is that because the kernel does not know about these threads, it cannot schedule multiple threads from the same process on multiple processors.

**Kernel Level Threads**

With KLTs, each thread is mapped to its own execution, and thread management is done by the kernel. These threads exist in the kernel space, and as such, have access to OS system calls. In contrast with ULTs, no thread management is done by the application.

The advantages of KLTs are that the kernel can simultaneously schedule multiple threads from the same process on multiple processors; if one thread in a process is blocked, for example, the kernel can schedule another thread of the same process. Hence, kernel routines can be multithreaded.

A problem with KLTs is the time it takes to transfer control from one thread to another within the same process. This is because the dispatcher needs to be executed every time the kernel schedules a new thread, which takes a non-trivial amount of time.

**Which is better?**

If a process is running on a single processor, and there are no system calls, then User-Level Threads are optimal; the dispatcher does not need to execute, and instead the Thread Library controls the scheduling. Because it is on a single processor, the kernel is not required to schedule threads across multiple processors.

If the process runs on multiple processors, the kernel is needed to schedule and manage the threads within the process across multiple processors, and Kernel-Level Threads are necessary. The Thread Library contained within the process cannot do this. Kernel-Level Threads can also be scheduled to handle OS system calls

1. Web servers are multithreaded environments. Explain the consequences of a Web server that is single-threaded.

Because they are single-threaded, these web servers cannot experience true concurrency; the single thread can only complete one task at a time, and each user has to wait for others’ requests to be completed before his/her request is processed.

It might be slower to handle larger requests because users could experience delays (less responsiveness), but because no threads have to spin up (which takes time), these servers can handle the entire queue of requests much more quickly than servers with multiple threads.

1. Consider an operating system that has only processes and doesn’t have kernel-level threads implemented. What advantages and disadvantages do you see in comparison with an operating system that has processes and kernel-level threads implemented?

A system that only has processes and doesn’t have kernel-level threads makes system calls by executing the dispatcher to spawn a whole new process for that system call, while timing out the current process.

The advantage of this configuration is that because the system call has its own process image, and thus its own heap, it will not interfere with the user-privilege process’s heap.

A disadvantage is that the kernel has to time-out the current process prematurely and execute the dispatcher to create a new process for the system call. If this configuration had kernel-level threads, the process could simply switch to kernel-level privilege and spawns a kernel stack, which takes significantly less time than the alternative.