# Homework #2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_

•Grading: 3 = correct

2 = almost

1 = an attempt

0 = nothing

•Score: Points / Possible

# (55 points) (Name) (Section)

**Chapter 3 – Processes**

**Chapter 4 – Threads**

|  |  |
| --- | --- |
| Questions: | Answers: |
| 1. (5 points) Consider a concurrent program with two processes, *p* and *q*, defined as follows: A, B, C, D, and E are arbitrary atomic (indivisible) statements. Assume that the main program (not shown) does a **parbegin** of the two processes (starts them both executing). Show all the possible interleaving of the execution of the preceding two processes.  **void p() void q()**  **{ A; { D;**  **B; E;**  **C; }**  **}** | **A B C D E**  **A B D C E**  **A B D E C**  **A D B C E**  **A D B E C**  **A D E B C**  **D E A B C**  **D A E B C**  **D A B E C**  **D A B C E** |
| 2. (10 points) Explain how the following applications would benefit from multithreading:   1. Web browser 2. Word processor 3. Multicore system 4. Operating system kernel | a. Web browser:Displays images or text.Retrieves data from the network.Listens for client requests.Service client requests.b. Word processor:Display graphicsRespond to keyboard inputsPerform spelling and grammar checksc. Multicore systems:CPU-intensive tasks in parallele. Operating System kernels:Manage devices.Manage memory.Interrupt handling. |
| 3. (10 points) Explain the benefits of multithreaded programming with respect to:   1. Responsiveness 2. Resource sharing 3. Economy 4. Scalability | a. Responsiveness: A program can continue to run even if part of it is blocked or is performing a lengthy operation.b. Resource sharing: Processes can only share resources through techniques such as shared memory or message passing. Threads share process resources by default.c. Economy: Allocating memory and resources for process creation is costly – more economical to create and context-switch threads.d. Scalability: As cores are added, threads may be run in parallel. |
| 4. (15 points) Elaborate on the programming challenges of multithreading applications.  a. Identifying units of work (tasks)  b. Balance  c. Data splitting  d. Data dependency  e. Testing and debugging | 1. Identifying tasks: How to find code within an application that can be divided into separate, concurrent tasks.2. Balance: Ideally tasks perform equal work of equal value.3. Data splitting: Effective caching requires equal data splitting as well.4. Data dependency: Tasks have to be synchronized to accommodate data dependencies.5. Testing and debugging: Inherently difficult to examine a program running in parallel on multiple cores – many different execution paths possible.Process creation is time consuming and resource intensive. |
| 5. (15 points) The output of the C program to the right is:  **tid=0, count=1**  **tid=1, count=1**  **tid=2, count=1**  **tid=3, count=1**  **tid=0, count=2**  **tid=1, count=2**  **tid=2, count=2**  **tid=3, count=2**  Answer the following questions:  a) Where in memory would you find the variable **i** (heap, kernel stack, thread stack)?  In the kernel stack.  b) Why is the variable **count** equal to 1 for the first four **printf**‘s in function **myThread** and then changes to 2?  A new local variable count is found in each thread’s stack.  c) What is the purpose of the **setjmp** function on line 025?  Capture a task context (thread stack) and return to kernel stack.  d) How many times is the function **myThread** called? What is the value of **code** when the function is called?  4 times. Code = 2.  e) What C statement is executed just after line 041 is executed for the 5th time?  Line 42. | **001 #include <setjmp.h>**  **002 #include <stdio.h>**  **003 #include <stdlib.h>**  **004 #include <ctype.h>**  **005**  **006 #define NUM\_THREADS 4**  **007 #define STACK\_SIZE (64\*1024)**  **008 #define STACK\_END (STACK\_SIZE/sizeof(int\*))**  **009 volatile void\* stack; // stack**  **010**  **011 int tid; // thread id**  **012 jmp\_buf thread[NUM\_THREADS]; // thread context**  **013 jmp\_buf kernel; // kernel context**  **014 void myThread(int); // thread function**  **015**  **016 int main()**  **017 {**  **018 int i, code;**  **019 for (tid = 0; tid < NUM\_THREADS; tid++)**  **020 {**  **021 if (setjmp(kernel) == 0)**  **022 {**  **023 stack = (int\*)malloc(STACK\_SIZE) + STACK\_END;**  **024 \_asm("movl \_stack,%esp"); // new stack pointer**  **025 if (!(code = setjmp(thread[tid]))) longjmp(kernel, 1);**  **026 myThread(tid);**  **027 }**  **028 }**  **029 for (i = 0; i < 8; i++)**  **030 {**  **031 tid = i % NUM\_THREADS; // select next thread**  **032 if (!(code = setjmp(kernel))) longjmp(thread[tid], 2);**  **033 }**  **034 }**  **035**  **036 void myThread(int tid)**  **037 {**  **038 int count = 0; // task iteration counter**  **039 while (1)**  **040 {**  **041 if (!setjmp(thread[tid])) longjmp(kernel, 3);**  **042 printf("\ntid=%d, count=%d", tid, ++count);**  **043 }**  **044 }** |