# 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; }**  **}** | abcde  abdce  abdec  adbce  adbec  adebc  dabce  dabec  daebc  deabc |
| 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 can have multiple tabs running at same timeb) word processor can run tasks in background (like word count, spell check, etc)c) multicore systems can use the extra cores to run different threadsd) can run several jobs concurrently |
| 3. (10 points) Explain the benefits of multithreaded programming with respect to:   1. Responsiveness 2. Resource sharing 3. Economy 4. Scalability | a) multithreaded programs generally work faster, and are therefore more responsiveb) depending on how the program is set up, a multithreaded program can do the same code on different bits of the data to compute a result much faster than a single threaded programc) to do the same thing a multithreaded program can do on many small cheap machines in a similar amount of time, a single threaded program would require a very large and expensive machined) a multithreaded program can just spawn more threads to increase scalability (within the capabilities of the machine upon which the program is being run) |
| 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 | a) not easy to divide a program into equal bits of workb) some threads may be given more work than othersc) data is not always able to be evenly split and distributed to all the threadsd) if a section of data depends on data being used in a different thread, they may get out of sync or cause deadlocke) a bug may not show up in the right place, so finding where it actually occurred is difficult |
| 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)?  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?  The variable count resides in the thread stack, so each thread has its own count.  c) What is the purpose of the **setjmp** function on line 025?  dunno, maybe to make sure that the thread being used is the right one?  d) How many times is the function **myThread** called? What is the value of **code** when the function is called?  Absolutely no idea  e) What C statement is executed just after line 041 is executed for the 5th time?  don’t undertand this question | **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 < 12; 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 }** |