CompSci 2SD3 Tutorial #10

TA: Jatin Chowdhary

DATE: April 12th, 2022

Announcements (1)

- Assignment #3 is marked
  - Marks should be out soon
  - If there is any issue, shoot me an email!
    - Or if you want to know your mark in advance.
- Assignment #4 won't be marked till next week
  - Not a pressing issue
    - Limited GO content on exam
      - *No GO programming*
  - These slides are more important

Announcements (2)

- Marking
  - Marks for assignment 1, 2, and 3 have been released
  - Marks for midterm 1 and 2 have been released
  - If there is any issue, send me an email!
    - This includes regrades – your mark will not decrease
- Midterm 2
  - Some guy in T01 got the highest mark (100%)
  - Plenty of people got 14/15 (in T03)
    - Virtually everyone lost a mark on the multiple choice

Other (1)

- What do you need?
  - Money
    - *Was not too effective for the midterm*
  - Motivation
    - *What motivates y'all?*
  - Food
    - *Should we order pizza right now?*
  - Answers
    - *Wanna see the final exam?*

Other (2)

- I think I've figured it out
  - What seems to work best is constant reminders
- Hence, expect hourly emails
  - Up until the exam
- I'll try to create more review stuff up until the exam
  - More questions/answers
  - *No promises*

Final Exam

- **This Thursday**
  - April 12th, 2022
    - *3 days from right now, literally*
- Start time = 7:30PM
- Duration = 2.5 hours
- Location = DBAC/IWC
- Outline can be found here
- *What exams do you have before 2SD3?*

Course Evaluation

- Take 3.3 mins to fill one out
- No need for 10 page reviews; a few lines is good enough
- Try to be honest
  - The evalulations are (mostly) anonymous
- Deadline is tonight!
- *Yes, we do read them!*

Any Questions?

Disclosure

- *I have **NOT** seen the final exam*

Thread Signaling (1)

- *This was on midterm 2*
- **Question:** If a signal is sent to a process with several threads, which of the threads receives the signal?
- **Options:**
  - A) All of the threads
  - B) Only the main thread (i.e. The thread with the lowest TID)
  - C) The thread that is just executing when the signal is received
  - D) None of the above
- **Answer:**

Thread Signaling (2)

- **Question:** If a signal is sent to a process with several threads, which of the threads receives the signal?
- **Explanation:**
  - A) In lecture Dr. Franek said *all* threads receive the signal. What he meant to say is *any* of the threads can receive the signal. This option is incorret.
  - B) This does not make sense. Why would the main thread only receive the signal? This is certainly possible, but it does not make sense.
  - C) The currently executing thread can receive the signal, but not always.
  - D) This is the correct answer. Any of the threads can receive the signal, even if it is blocked or sleeping.

Thread Signaling (3)

- **Question:** If a signal is sent to a process with several threads, which of the threads receives the signal?
- **Explanation:**
  - Signal handling with threads needs to be handled by you, the programmer. There are various approaches you can take, for example:
    - Dedicate a single thread for handling signals and communicating with other threads
    - Force the currently executing thread to handle the signal
    - Etc.
  - The main takeaway is that if you send a signal to a process, you cannot determine which thread handles the signal, because it is implementation specific.
    - In other words, you have to program it.

Question #2

- **Question:** Refer to *question2*; Which scenario will we most likely see.
  - Scenario A:
    - Thread 22220, hey
    - Thread 22220, bye
    - Thread 22222, hey
    - Thread 22222, bye
    - Terminating
  - Scenario B:
    - Thread 22220, hey
    - Thread 22220, bye
    - Terminating

Question #2

- **Options:**
  - A) Scenario A, because the critical section of doit() starts with pthread_mutex_lock(&flock); and ends with pthread_exit(NULL);. Thus each thread is “protected” during the “conversation” with the user.
  - B) Scenario B, since the critical section of doit() starts with pthread_mutex_lock(&flock); and ends with pthread_exit(NULL); and thus the mutex flock is never unlocked. Thus the first thread will lock flock and the second thread will wait forever.
  - C) Scenario A, because the critical section of doit() starts with pthread_mutex_lock(&flock); and ends with pthread_mutex_unlock(NULL);. Thus each thread is “protected” during the “conversation” with the user.
  - D) It cannot be determined, it depends on the timing of events. If the first thread terminates before the second thread tries to lock flock, everything will be OK and we will see Scenario A, otherwise we will see Scenario B.
  - E) Scenario A, because the critical section of doit() starts with pthread_mutex_lock(&flock); and ends with pthread_mutex_unlock(&flock);. Thus each thread is “protected” during the “conversation” with the user.
- **Answer:**

Question #3

- **Question:** Refer to *question3*; Which scenario will we most likely see.
  - Scenario A:
    - Thread 22220, hey
    - Thread 22220, bye
    - Thread 22222, hey
    - Thread 22222, bye
    - Terminating
  - Scenario B:
    - Thread 22220, hey
    - Thread 22220, bye
    - Terminating
  - Scenario C:
    - Thread 22222, hey
    - Thread 22222, bye

Question #3

- **Options:**
  - A) Scenario A, because the the variable *areWeThereYet* is local to each thread. Hence both threads get to execute without interferring with each other.
  - B) Scenario B, because the variable *areWeThereYet* is global. So thread 2 will execute first and change the value of areWeThereYet, which will prevent thread 3 from running.
  - C) Scenario C, because thread 3 will execute first but it won't be able to finish due to not being able to release the mutex lock. The execution is blocked.
  - D) Order of execution cannot be determined. We have no control over the scheduler and how it determines which process/thread gets to execute.
  - E) Both Scenario A and B are possible.
  - F) None of the above
- **Answer:**

Question #4

- **Question:** Which of the following format specifier is incorrect for *pthread_self()*?
- **Options:**
  - A) %d (signed int)
  - B) %u (unsiged int)
  - C) %s (string)
  - D) %lu (long unsigned int)
  - E) All of the above
  - F) None of the above
  - G) A and C only
- **Answer:**

Question #5

- **Question:** Is it acceptable to use %d as a format specifier for thread IDs (i.e. pthread_self())? What are the implications?
- **Hint:**
  - Think about the size/range of %d
  - How many thread IDs are present in a system?
  - How big should a thread ID be?

Question #5

- **Question:** Is it acceptable to use %d as a format specifier for thread IDs (i.e. pthread_self())? What are the implications?
- **Answer:**
  - Normally, it is not a good idea to use %d as a format specifier for pthread_self(). This is because %d is for signed integers. The range allowed via %d is not large enough to represent thread IDs. However, this depends on the system. Some (embedded) systems might have small thread ID numbers, so using %d is acceptable. But for most (mainstream) systems, %d is not acceptable. Even though it will not crash the program, the thread IDs can come out as negative, because %d is not large enough.

Question #6

- **Question:** List and briefly explain the Coffman conditions that cause deadlock?
- **Hint:**
  - Mut… Ex…
  - Hold and W…
  - No Preemp…
  - Circular W…

Question #6

- **Question:** List and briefly explain the Coffman conditions that cause deadlock?
- **Answer:**
  - A deadlock occurs if the four Coffman conditions hold true. However, these conditions are not mutually exclusive:
    - *Mutual Exclusion:* There's a resource that can only be held by one process at a time (i.e. Non-shareable mode). Only one process can use that resource at any given instant of time.
    - *Hold & Wait:* A process can hold multiple resources and still request more resources from other processes that are holding them.
    - *No Preemption:* A resource cannot be preempted (taken) from another process by force. Processes can only voluntarily release resources.
    - *Circular Wait:* A process is waiting for a resource held by another process. That process is waiting for a different resource that is held by another process. And so on…
      - i.e. P1 waits for P2. P2 is waiting for P3. P3 is waiting for P1
        - P1 → P2 → P3 → P1 → P2 → …

Question #7

- **Question:** Give an example of a deadlock detection and recovery system. Elaborate on the advantages and disadvantages of your system.
- **Hint:**
  - Many possible answers
  - Think of a guard that *watches* over the system

Question #7

- **Question:** Give an example of a deadlock detection and recovery system. Elaborate on the advantages and disadvantages of your system.
- **Answer:**
  - The system can have a timer – software based – that monitors how long a resource has been in possession by a process. If a significant amount of time has passed, the timer can preempt (forcefully release) the resource that is occupied by the deadlocked process.
    - This is called Watchdog Timer.

Question #7

- **Question:** Give an example of a deadlock detection and recovery system. Elaborate on the advantages and disadvantages of your system.
- **Answer:**
  - Advantages
    - Watchdog timers are quite flexible and can be used to correct many different kinds of errors/failures, including deadlock.
      - A single WDT can be used to safeguard an entire system
    - Easier to implement than model checking and proving correctness
      - State explosion makes proving software very difficult
  - Disadvantages
    - Require more resources
    - Hard to determine the interval
      - i.e. How long until preemption?

Question #8

- **Question:** Which of the following scheduling algorithm is provably optimal? Meaning, it gives the minimum average waiting time for a given set of processes.
- **Options:**
  - A) Round Robin
  - B) First Come First Serve
  - C) Priority
  - D) Shortest Job First
  - E) None of the above
- **Answer:**

Question #9

- **Question:** Which of the following scheduling algorithm cannot be preemptive?
- **Options:**
  - A) Round Robin
  - B) First Come First Serve
  - C) Priority
  - D) Shortest Job First
  - E) None of the above
- **Answer:**

Question #10

- **Question:** Which of the following scheduling algorithm typically has the longest/largest average waiting time?
- **Options:**
  - A) Round Robin
  - B) First Come First Serve
  - C) Priority
  - D) Shortest Job First
  - E) None of the above
- **Answer:**

Question #11

- **Question:** In the Shortest Job First (SJF) scheduling algorithm, if the next 2 processes/jobs have the same (CPU) burst time, which scheduling algorithm is used to determine the next job?
- **Options:**
  - A) Round Robin
  - B) First Come First Serve
  - C) Priority
  - D) Shortest Job First
  - E) None of the above
- **Answer:**

Question #12

- **Question:** The Shortest Job First (SJF) scheduling algorithm must be:
- **Options:**
  - A) Can only be preemptive
  - B) Can only be non-preemptive
  - C) Does not matter; both preemptive and non-preemptive are acceptable
  - D) All of the above
  - E) None of the above
- **Answer:**

Question #13

- **Question:** A system containing four resources of the same type that are shared by three threads, each of which needs at least two resources, is in a deadlock state.
- **Options:**
  - A) True
  - B) False
  - C) Cannot be determined
  - D) Not enough information
  - E) None of the above
- **Answer:**

Question #14

- **Question:** Is it possible to have deadlock with a single mutex? Explain why or why not?
- **Hint:**
  - There are 2 sides to this answer
  - Pick a side and explain

Question #14

- **Question:** Is it possible to have deadlock with a single mutex? Explain why or why not?
- **Answer:**
  - Yes and no; depends on the situation. If the single mutex is locked and unlocked appropriately, then there is no deadlock. If a thread/process fails, or forgets, to release the mutex, then there can be deadlock. This is called *Hold & Wait*.

Question #15

- **Question:** What is the difference between an atomic variable and mutex lock?
- **Hint:**
  - What is a mutex lock?
  - What is an atomic variable?
  - Think about where you would use each one

Question #15

- **Question:** What is the difference between an atomic variable and mutex lock?
- **Answer:**
  - A mutex lock protects a critical section. The critical section can span many lines of code or protect a single variable. In short, mutexes are used to perform mutually exclusive actions. On the other hand, an atomic variable performs an atomic operation, which is a single operation that is mutually exclusive.
  - For example: mutexes can be used to protect writing to a file, modifying global variables, etc. On the other hand, an atomic variable can only protect a (global) variable via atomicity (the operation becomes indivisible).

Question #16

- **Question:** Which of the following cannot cause deadlock?
- **Options:**
  - A) Not releasing the mutex lock, which causes the thread to have exclusive access, there by preventing a deadlock
  - B) If two or more mutex locks are used, the order in which the locks are acquired are different for each thread, which prevents deadlock because each thread can acquire a different lock
  - C) If there is a circular wait where P1 is waiting on P2, and P2 is waiting on P3, and P3 is waiting on P1.
  - D) All of the above
  - E) None of the above
- **Answer:**

Question #17

- **Question:** Which of the following is NOT a system call?
- **Options:**
  - A) *fork()*
  - B) *wait()*
  - C) *pthread_create()*
  - D) *pthread_exit()*
  - E) Two of the above
  - F) None of the above
  - G) All of the above
- **Answer:**

Question #18

- **Question:** Which of the following scheduling algorithms could result in starvation?
- **Options:**
  - A) First Come First Serve
  - B) Priority Scheduling
  - C) Shortest Job First
  - D) Round Robin
  - E) All non-preemption algorithms
  - F) A and B
  - G) C and D
  - H) B and C
  - I) None of the above
- **Answer:**

Question #19

- **Question:** Which of the following scheduling algorithms could result in starvation?
- **Options:**
  - A) First Come First Serve
  - B) Priority Scheduling
  - C) Shortest Job First
  - D) Round Robin
  - E) All non-preemption algorithms
  - F) A and B
  - G) C and D
  - H) B and C
  - I) None of the above
- **Answer:**

Question #20

- **Question:** What is the difference between concurrency and parallelism?
- **Answer:**
  - Concurrency is when two or more tasks make progress and complete in overlapping time periods (take turns).
    - i.e. Cutting several vegetables with a single knife
    - i.e. Two or more lines of customers ordering food from a single cashier. Customers take turns ordering food.
  - Parallelism is when two or more tasks/jobs run at the same time (simultaneously).
    - i.e. Cutting several vegetables, using both hands, with a knife in each hand, and each knife cuts a different vegetable
    - i.e. Two or more lines of customers ordering food from two or more cashiers. Each cashier handles its own line.

Question #21

- **Question:** Is it possible to have concurrency but not parallelism?
- **Options:**
  - A) No, because parallelism is a subset of concurrency
  - B) Yes; the two have little to do with each other. The concepts are separate ideas
  - C) Only under select circumstances with very specific hardware
  - D) Not enough information provided to answer this question
  - E) None of the above
- **Answer:**

Question #21

- **Question:** Is it possible to have concurrency but not parallelism?
- **Answer:**
  - It is possible to have concurrency but not parallelism. Consider a single-core CPU and a multi-threaded application. The threads can take turns executing on the CPU. This way, all threads can make progress ‘‘at the same time’’, and reach completion.
    - Note: ‘‘at the same time’’ implies that calculations are being performed so quickly that progress appears to be simultaneous (to the human eye).

Question #22

- **Question:** Original versions of Apple's mobile operating system, iOS, provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system
- **Hints:**
  - Programming/debugging
  - Security
  - Deadlock

Question #22

- **Question:** Original versions of Apple's mobile operating system, iOS, provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system
- **Answer:**
  - *Complex:* The added complexity of concurrency, makes it much (much) harder to (properly) design, program, test, and debug a concurrent system; when compared to non-concurrent systems. It takes more time, and money, to create a successful concurrent system.
  - *Security:* Concurrency introduces new security holes to a system that can be maliciously exploited by an attacker. If concurrency isn't properly implemented it can lead to race condition bugs, memory corruption, buffer/stack overflow, etc.
  - Deadlock: This is a major problem introduced by concurrent programming. Deadlocks lead to a variety of problems such as…
    - *Homework; think about everything you have learned so far.*

Question #23

- **Question:** Why are concurrency and paralleism important?
- **Hint:**
  - Moore's Law
  - Performance
    - Efficiency
    - Speed
  - *Think about the advantages of each!*

Question #23

- **Question:** Why are concurrency and paralleism important?
- **Answer:**
  - *Homework*
    - *Use the hints provided*

Question #24

- **Question:** Is it possible for a non-concurrency system to experience deadlock?
- **Options:**
  - A) Yes
  - B) No
  - C) Maybe
  - D) Depends
- **Answer:**

Still More To Review

- GO
- IPC
- Scheduling
- Deadlock
- Signals
- Processes
- System Calls
- System Stack

THE END