CompSci 2SD3 Tutorial #7

TA: Jatin Chowdhary

DATE: March 15th, 2022

Announcements (1)

- CompSci 2SD3 **is** a coding course
  - Assignments are 100% coding
  - Tests are 85% coding
  - Lectures revolve around coding
- New structure:
  - First hour: Learning the basics *(i.e. The merit)*
    - *2 minute (market) break*
  - Second hour: Learning how to do it *(i.e. Code)*
- New expectations:
  - Lots of hand holding, spoon feeding, etc.
  - Understanding every tiny little granular detail
  - Lots of participation vis a vis **this**
    - *I'll make my own program later*

Announcements (2)

- Wanted to bring food – they said I can't because COVID
  - Instead, I'll do you one better: *Cold hard cash*
- Whoever gets the highest mark in the class gets $200, cash
  - But for the 2nd midterm, whoever gets the highest mark gets $100, cash
  - Note: You must be in **T03** to be eligible for the prize
- Motivation:
  - Get more effort out of you
  - Raise the stakes

Announcements (3)

- Quick note on Assignements:
  - Email the TA that **YOU** are assigned to for any questions
  - We want to avoid a conflict where TA-1 said, ‘‘The sky is blue’’, and TA-2 said, ‘‘The sky is green’’.
    - Had this issue last semester in 2GA3
      - Just ask Kai *(but don’t ask him for investment advice)*
- Honesty
  - If you do not understand something, then proudly say, ‘‘I don’t get it’’.
    - *Get comfortable shouting this from rooftops*

Honesty

- There is nothing wrong with:
  - Not knowing the answer
  - Answering a question incorrectly
  - Admitting your fault(s)
- I still don’t know anything about the market
  - I’ve been trading for months now, and I still don’t know anything

Any Questions

Cash Prize

- Highest mark on the midterm (in the entire class)
  - $100 CASH
- Highest mark in the course (in the entire class)
  - $200 CASH
- Rules:
  - Must be in my section: **T03**
  - If the mark is curved, all bets are off
    - You need to get a 100% before the curve
    - *I don't believe in curves*

Outline

- Midterm #2 Review
  - Coding questions
    - Related to threads and fork
  - Theory questions
    - Race conditions, threads, processes, etc.
- Anything else you want me to cover
  - Assignment #2
  - Lecture content
- Note: Slides/code will be posted

Midterm #2 (Deprecated)

- ***Old prediction for Midterm #2***
- Midterm #2
  - 1. Mix of *fork()* and *pthreads*
    - i.e. Fork a process and create multiple threads
    - i.e. Theory questions about threads
  - 2. All about pthread library (and threads)
    - i.e. Difficult thread related coding questions that involves things like attributes, cancellation, synchronicity, stack, etc.
  - Waiting for sample midterm
    - *No idea if Dr. Franek will post one*

Midterm #2 (Official)

- The following was posted by Dr. Franek
  - In grey, are my predictions
- Midterm #2
  - 1. Logging functions from log.c
    - *i.e. msg(), msg_exit(), sys_exit(), etc.*
  - 2. Scheduling, deadlocks
    - *i.e. Understand the concepts*
  - 3. Linux processes, threads
    - *i.e. Know the difference between the two, advantages and disadvantages to each, and how to implement them in practice.*
  - 4. Functions like: pthread_create(), pthread_join(), pthread_exit()
    - *i.e. Lots of examples in these slides. Basically, know what each function does, the arguments it takes, return type, etc. It would be a good idea to read the man pages for ``pthread`` and write down important information to your cheat sheet.*
  - 5. Mutexes, definition and initialization, locking and unlocking
    - *i.e. Tutorial #5 has a lot of information, examples, etc. about mutexes.*
  - 6. Signals to threads
    - *i.e. Similar to questions on midterm #1 and the sample midterm. However, the difference is that signals are sent to and handled by threads.*

Any Questions

Question #1

- Refer to *question1.c* and answer the following questions:
  - A) How many times is "*I am a thread*" printed?
  - B) Why doesn't the threaded function, *runner*, execute?
  - C) How would you fix the problem from *(B)*?
  - D) Will adding *sleep(1)*, before *return 0*, fix the issue from *(B)*?
  - E) Is the fix from *(E)* an acceptable solution? Why or why not?
  - F) How many unique processes are created?
  - G) How many threads are created?
  - H) Draw the corresponding process/thread tree diagram

Answer #1

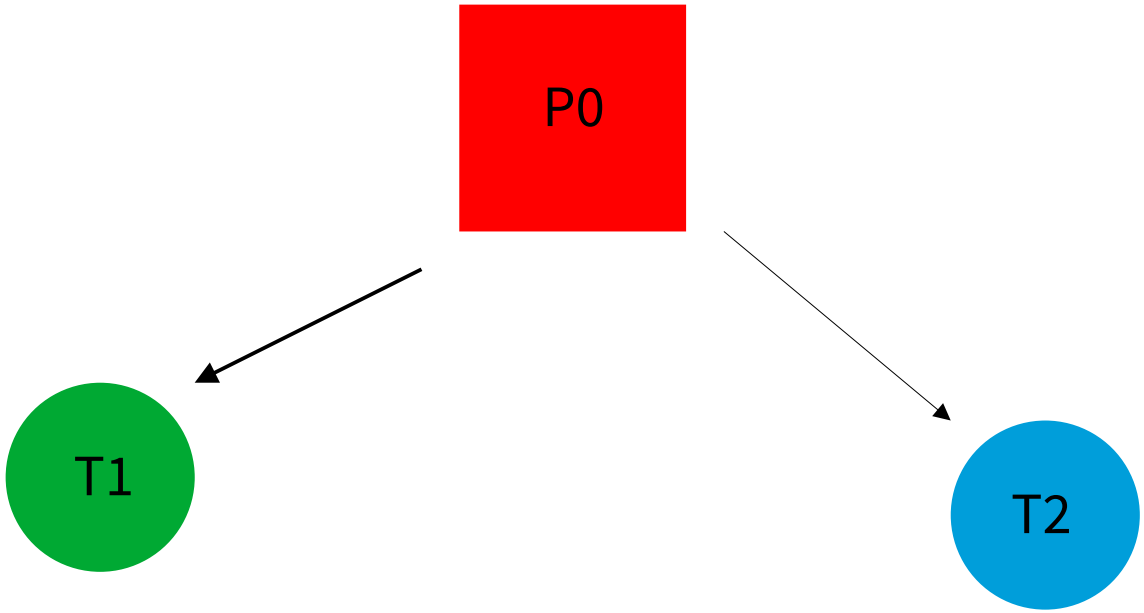
- Sample answers to Question #1, based on *question1.c*:
  - A) Technically, it should be printed 2 times. However, since there is no *pthread_join()*, it ends up not being printed at all. Rarely, it is printed once, but the general answer is that it is not printed at all.
  - B) The threaded function does not run because the process, which we will call the main thread, terminates before the other threads. Once it terminates, the other created threads also terminate. Rarely, the threaded function runs and something is printed to the terminal. It is also possible that the threaded function does execute, but nothing is printed because the threads terminte prematurely.
  - C) We can fix the problem by adding *pthread_join()* after the threads are created, or right before *return 0*. Also, it is a good idea to use a different *tid* variable for each created thread.
  - D) Yes, it does fix the issue, but it is not an acceptable solution.
  - E) Adding *sleep(1)* is not an acceptable solution because the main thread ‘‘waits’’ for one second. If the threads executed computationally intensive work that lasted several seconds, then they would be prematurely terminated.
  - F) How many unique processes are created?
  - G) How many threads are created?
  - H) Draw the corresponding process/thread tree diagram

Answer #1

- Sample answers to Question #1, based on *question1.c*:
  - F) How many unique processes are created?
    - No unique processes are created, apart from the parent process. We can show this by:
      - 1 + 0 → (Parent + Child)
  - G) How many threads are created?
    - In total, 2 threads are created.
  - H) Draw the corresponding process/thread tree diagram
    - *Next slide*

Answer #1

- Sample answers to Question #1, based on *question1.c*:
  - H) Draw the corresponding process/thread tree diagram



Any Questions

Question #2

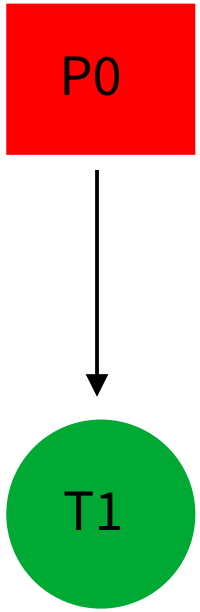
- Refer to *question2.c* and answer the following questions:
  - A) Identify the error(s) in *question2.c*
  - B) How would you fix the error identified in *(A)*?
  - C) What improvement(s) can you make to the code?
  - D) How many unique processes are created?
  - E) How many threads are created?
  - F) Draw the corresponding process/thread tree diagram

Answers #2

- Sample answers to Question #2, based on *question2.c*:
  - A) The error is in *pthread_join()*. The wrong thread ID is used.
  - B) To fix the error, simply change *tidl* to *tid1*. This is because *tid1* is used in *pthread_create()*, and this is where the ID of the thread is stored.
  - C) To improve the code, we can put *pthread_join()* inside an if-statement, similar to *pthread_create()*. This is because it is good practice to always check the return value to make sure the function terminated successfully.
  - D) How many unique processes are created?
    - No unique processes are created, apart from the parent process. We can show this by:
      - 1 + 0 → (Parent + Child)
  - E) How many threads are created?
    - In total, 1 thread is created
  - F) Draw the corresponding process/thread tree diagram
    - *Next slide*

Answer #2

- Sample answers to Question #2, based on *question2.c*:
  - H) Draw the corresponding process/thread tree diagram



Any Questions

Question #3

- Refer to *question3.c* and answer the following questions:
  - A) How many times is ‘‘I am a thread’’ printed?
  - B) How many times is ‘‘I am a thread too’’ printed?
  - B) Explain the reasonsing behind your answer from (A) and (B)?
  - C) How would you fix the issue identified in *(B)*?
  - D) After fixing the issue, answer (A) and (B) again.
  - D) How many unique processes are created?
  - E) How many threads are created?
  - F) Draw the corresponding process/thread tree diagram

Answer #3

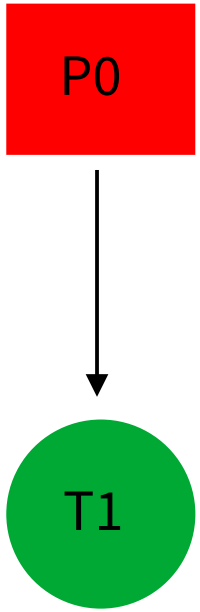
- Sample answers to Question #3, based on *question3.c*:
  - A) ‘‘*I am a thread*’’ is printed 0 times.
  - B) ‘‘*I am a thread too*’’ is printed 0 times.
  - B) The statements are not printed because the thread is terminated via *pthread_exit(0)* before *printf()* is executed.
  - C) The simple fix is to move *pthread_exit(0)* below the *printf()* statements. This way, the thread exits after the statements are printed. Also, *pthread_join()* is already executed after *pthread_create()*, so there is no issue with the process terminating before the threads.
  - D) The statement ‘*I am a thread*’’ is printed 5 times, and the statement ‘‘*I am a thread too*’’ is printed 1 time. The latter is only printed once because it is outside of the for loop.
  - D) How many unique processes are created?
  - E) How many threads are created?
  - F) Draw the corresponding process/thread tree diagram

Answer #3

- Sample answers to Question #3, based on *question3.c*:
  - D) How many unique processes are created?
    - No unique processes are created, apart from the parent process. We can show this by:
      - 1 + 0 → (Parent + Child)
  - E) How many threads are created?
    - Only 1 thread is created
  - F) Draw the corresponding process/thread tree diagram
    - *Next slide*

Answer #3

- Sample answers to Question #3, based on *question3.c*:
  - H) Draw the corresponding process/thread tree diagram



Any Questions

Question #4

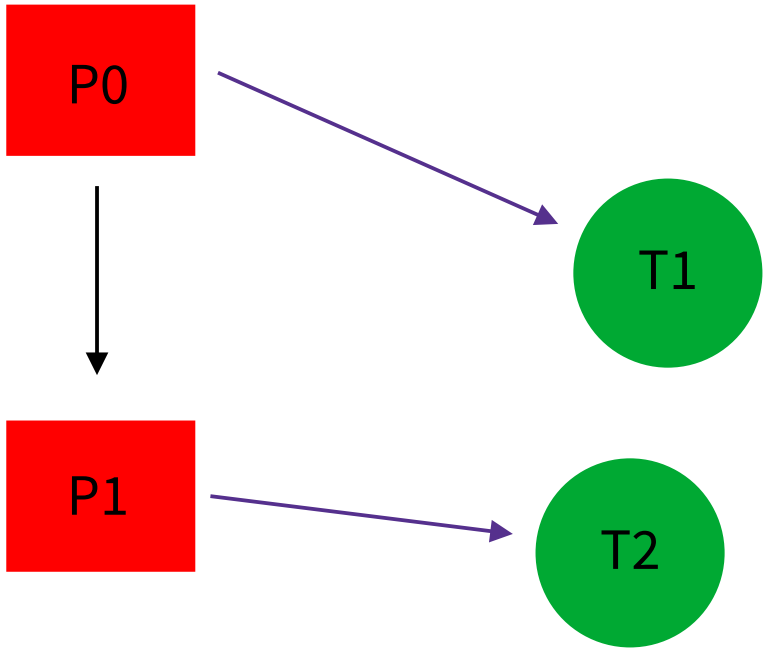
- Refer to *question4.c* and answer the following questions:
  - A) How many times is ‘‘Start here’’ printed?
  - B) How many times is ‘‘I am a thread’’ printed?
  - C) How many times is ‘‘End here’’ printed?
  - D) If the thread fails to create, is it acceptable (as per Unix convention) to return 0?
    - Will the program fail to run/compile if you return 0?
  - E) How many unique processes are created?
  - F) How many threads are created?
  - G) Draw the corresponding process/thread tree diagram

Answer #4

- Sample answers to Question #4, based on *question4.c*:
  - A) "*Start here*" is printed once.
  - B) "*I am a thread*" is printed 2 times.
  - C) "*End here*" is printed 2 times.
  - D) If the thread fails to create, it is NOT acceptable to return 0, as per Unix convention. This is because 0 means success and anything else means error/failure. The program will compile/run regardless of whether you put down 0 or 1.
  - E) How many unique processes are created?
    - A single child process (P1) is created by the parent process (P0). We can show this by:
      - 1 + 1 → (Parent + Child)
  - F) How many threads are created?
    - In total, 2 threads are created. P0 creates a thread and P1 creates a thread.

Answer #3

- Sample answers to Question #4, based on *question4.c*:
  - H) Draw the corresponding process/thread tree diagram



Any Questions

Question #5

- For this question, you can build on your answer from before. Refer to *question5.c* and answer the following questions:
  - A) How many times is "*Start here*" printed?
  - B) How many times is "*I am a thread*" printed?
  - C) How many times is "*End here*" printed?
  - D) Provide a sample output of *question5.c*
  - E) How many unique processes are created?
  - F) How many threads are created?
  - G) Draw the corresponding process/thread tree diagram

Answer #5

- Sample answers to Question #5, based on *question5.c*:
  - A) "*Start here*" is printed once.
  - B) "*I am a thread*" is printed 3 times.
  - C) "*End here*" is printed 2 times.
  - D) A sample output is:

    > Start here

    > I am a thread with ID: XXX | My parents ID is: AAA

    > End here

    > I am a thread with ID: XXX | My parents ID is: BBB

    > I am a thread with ID: XXX | My parents ID is: BBB

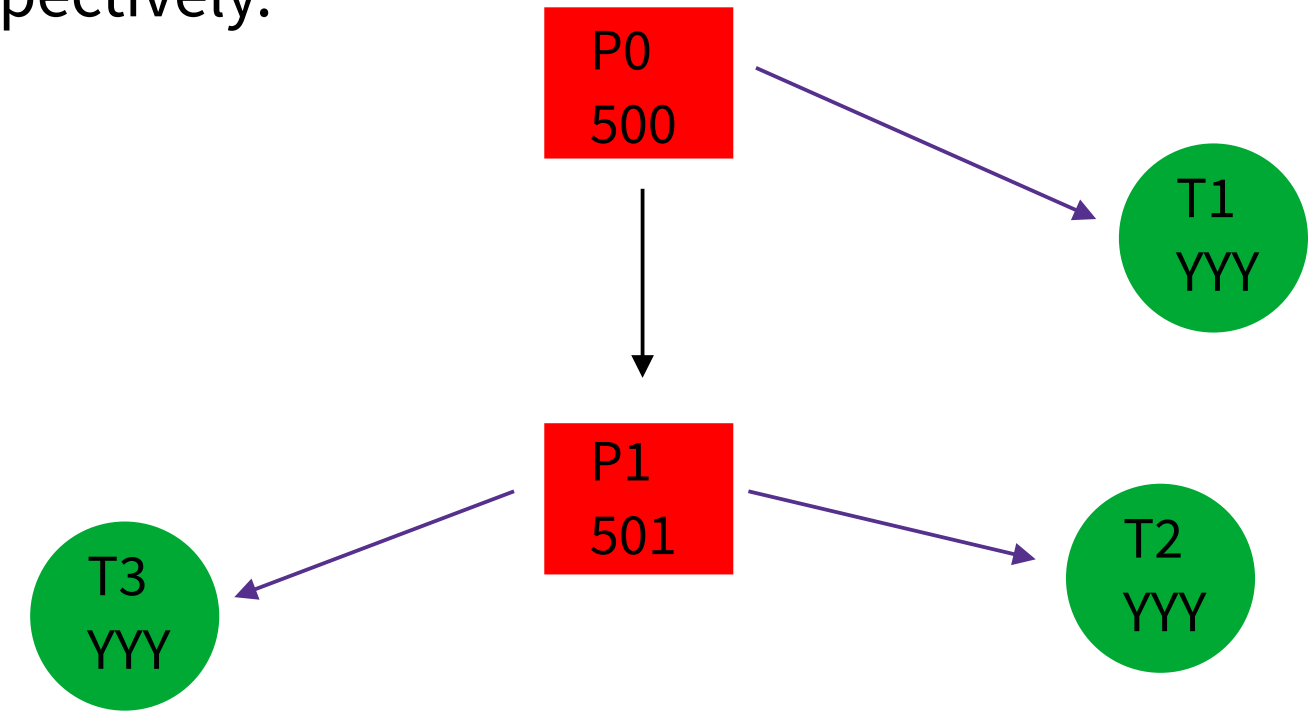
    > End here

Answer #5

- Sample answers to Question #5, based on *question5.c*:
  - E) How many unique processes are created?
    - A single child process (P1) is created by the parent process (P0). We can show this by:
      - 1 + 1 → (Parent + Child)
  - F) How many threads are created?
    - In total, 3 threads are created. P0 creates a single thread, and P1 creates 2 threads.
  - G) Draw the corresponding process/thread tree diagram. Provide the PID and TID of each process and thread, respectively.
    - *Next slide*

Answer #5

- Sample answers to Question #5, based on *question5.c*:
  - G) Draw the corresponding process/thread tree diagram. Provide the PID and TID of each process and thread, respectively.



Note

- *pthread_self()* is NOT the same as *gettid()*
  - *gettid()* gets the thread ID assigned by the system (or the kernel)
  - pthread_self() returns the calling thread's ID
  - The two are different because *gettid()* returns a unique number for every single thread, where as *pthread_self()* does not.
- *For the midterm, I'm not sure which one Dr. Franek will use, but make sure you understand the difference.*

Any Questions

Question #6

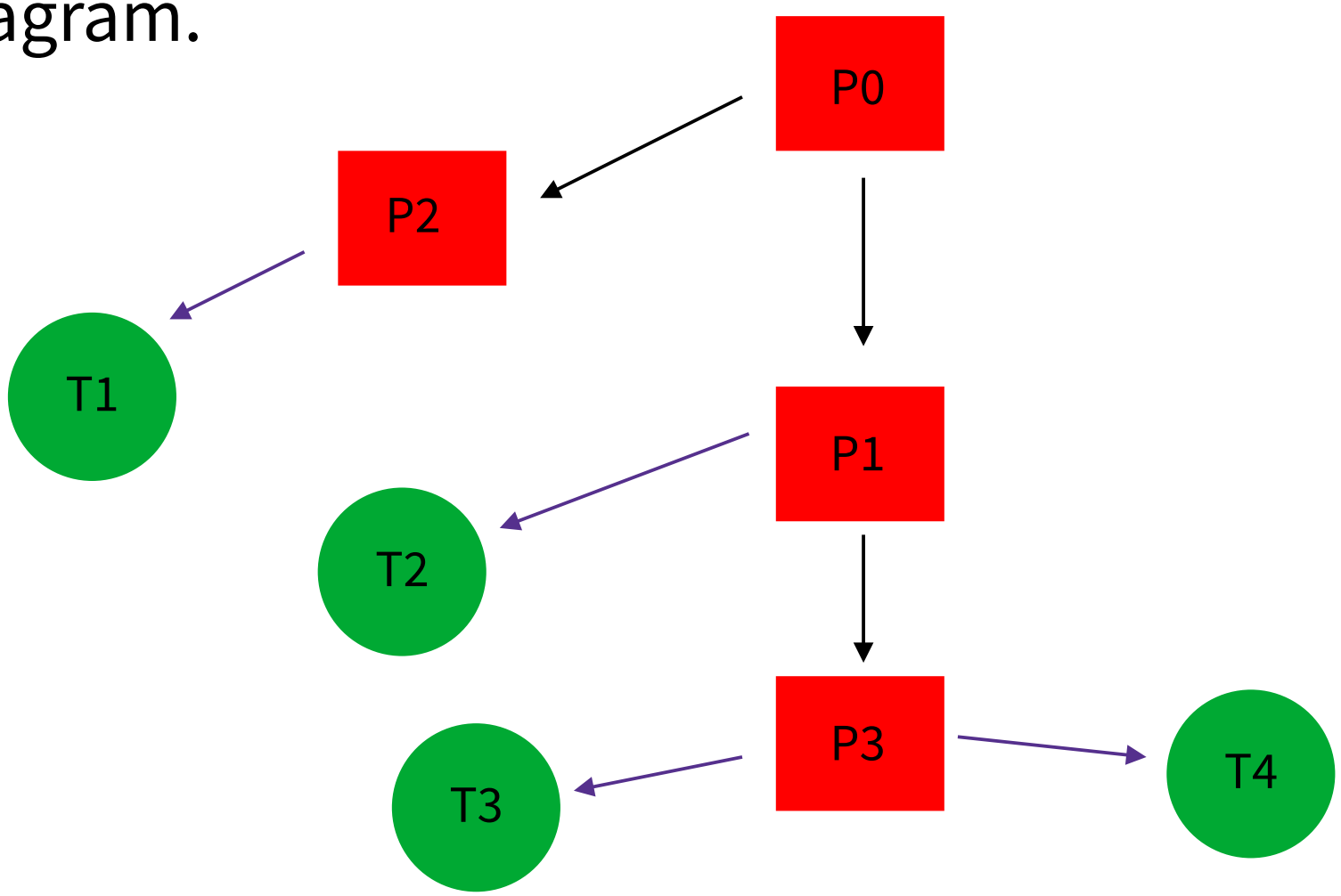
- Refer to *question6.c* and answer the following questions:
  - A) How many unique processes are created?
  - B) How many threads are created?
  - C) Draw the corresponding process/thread tree diagram

Answer #6

- Sample solution to Question #6, based on *question6.c*:
  - A) How many unique processes are created?
    - In total, 3 unique processes are created. We can show this by:
      - 1 + 3 → (Parent + Child)
  - B) How many threads are created?
    - In total, 4 threads are created. P1 and P2 create a thread, and P3 creates 2 threads.
  - C) Draw the corresponding process/thread tree diagram
    - *Next slide*

Answer #6

- G) Draw the corresponding process/thread tree diagram.



Any Questions

Question #7

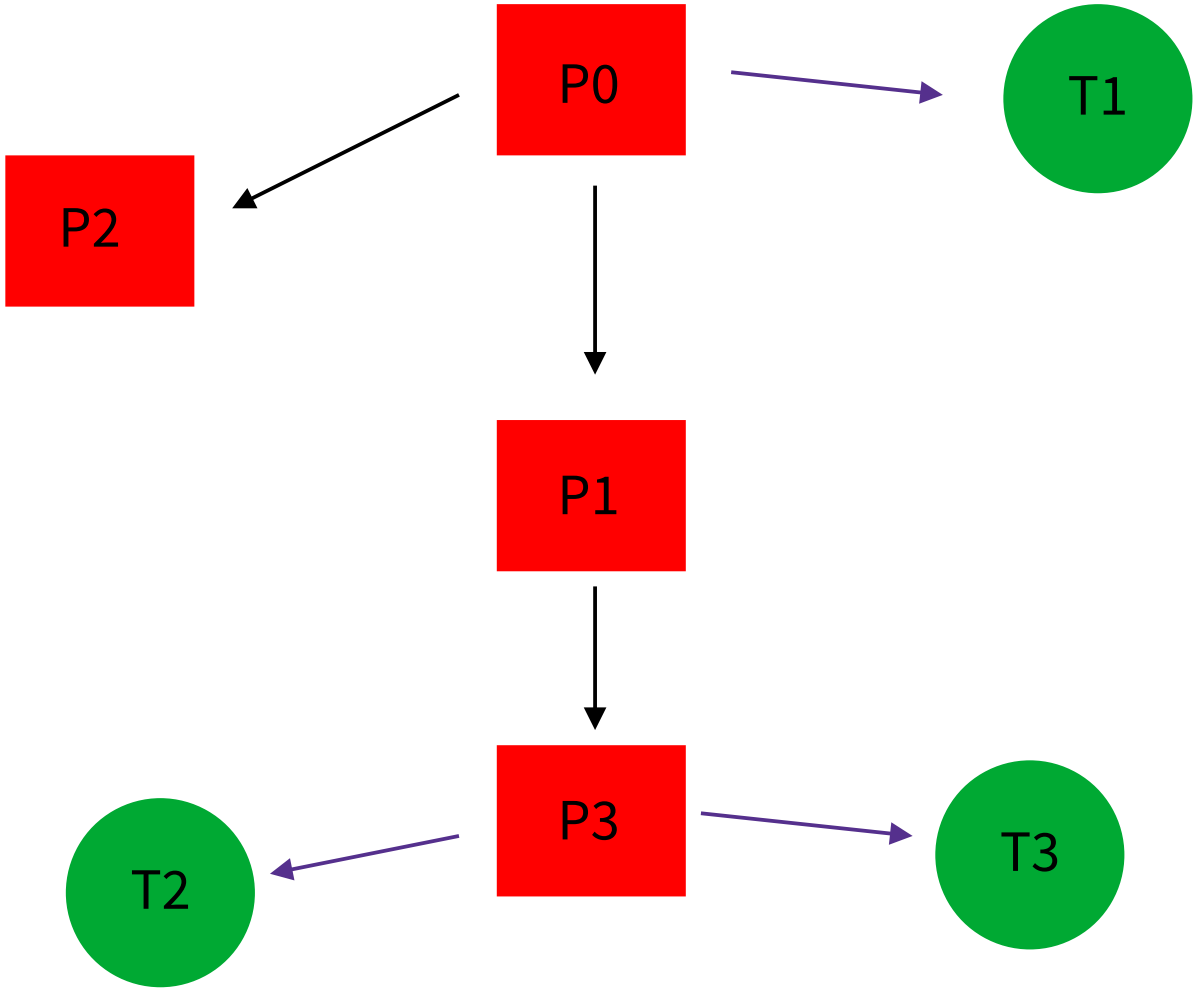
- Refer to *question7.c* and answer the following questions:
  - A) How many unique processes are created?
  - B) How many threads are created?
  - C) Draw the corresponding process/thread tree diagram

Question #7

- Sample answers to Question #7, based on *question7.c*:
  - A) How many unique processes are created?
    - In total, 3 unique processes are created. We can show this by:
      - 1 + 3 → (Parent + Child)
  - B) How many threads are created?
    - In total, 3 threads are created. P0 creates a single thread, and P3 creates 2 threads.
  - C) Draw the corresponding process/thread tree diagram
    - *Next slide*

Answer #7

- G) Draw the corresponding process/thread tree diagram.



Any Questions

Question #8

- **Question:** Assume you are a hacker. Explain the advantages and disadvantages of exploiting a race condition.
- **Hint:**
  - What is a race condition?
  - How reliable is a race condition?
  - How easy is it to catch/find race conditions?

Answer #8

- **Question:** Assume you are a hacker. Explain the advantages and disadvantages of exploiting a race condition.
- **Answer:**
  - The disadvantage of exploiting a race condition is its low success rate. Since things like order of execution is not guaranteed, triggering the bug can be unsuccessful at times. In addition, extra work may be required to optimize the exploit to take advantage of the vulnerability.
  - The advantage of race conditions is that they can slip through testing and debugging. Since race conditions seldomly show up, developers can easily miss them, causing the bugs to be present in official builds released to the public. As an attacker, this is advantageous because race conditions will always be present.
  - *Note: The best kind of vulnerability to exploit are ones that provide a 100% success rate. For example, buffer overflows.*

Question #9

- **Question:** Explain the implications of letting the user release mutex locks with the click of a button. Assume the program has been deadlocked.
- **Precursor:**
  - What are race conditions?
  - What are mutex locks?
  - What is a deadlock?

Answer #9

- **Question:** Explain the implications of letting the user release mutex locks with the click of a button. Assume the program has been deadlocked.
- **Answer:**
  - If a program is deadlocked, letting the user press a button (i.e. *Refresh*) to release the locks can be a good and bad thing. The main issue is the amount of control the user is given. For instance, if the program is frozen/deadlocked, the user can manually release the locks and restart the program. However, it may be difficult for some users to determine if/when the program is deadlocked. This can lead to the user constantly resetting the program under the false pretense that the system is frozen. In theory this practice sounds viable but in reality it is not applicable.

Question #10

- **Question:** Explain the difference between processes and threads. Briefly list the advantages and disadvantages of each one.
- **Hint:**
  - Think about:
    - CPU time (i.e. Cost to create each one)
      - Heavy VS. Light
    - Memory (i.e. What is shared?)
      - Register, Stack, Etc.

Answer #10

- **Question:** Explain the difference between processes and threads. Briefly list the advantages and disadvantages of each one.
- **Answer:**
  - A process refers to a program, where as a thread refers to a segment of that program. A process can have multiple threads. But threads cannot have processes.
  - Processes run independently of one another. This includes their memory space. A change made in one process does not affect another process. Apart from shared memory segments, processes do not share code, variables, heap, stack, registers, etc. In other words, processes and their memory are isolated. On the other hand, threads are part of a process. Threads in a process share code, file, and data. An alteration to a global variable is visible to all threads in the same process. However, threads do not share registers and stack space.
  - Processes are heavy-weight and threads are light-weight. Processes take more time to create, perform a context switch, and terminate. On the other hand, threads take less time than processes for creation, context switching, and termination.

Question #11

- **Question:** Give a real world example of where you would use threads over processes, and vice versa.
- **Hint:**
  - Use your answer from Question #10.

Answer #11

- **Question:** Give a real world example of where you would use threads over processes, and vice versa.
- **Answer:**
  - Processes are used when programs need to run independently of one another. For example, if you have multiple browsers on your computer, then each browser (i.e. Chrome, Firefox, Safari, etc.) should run as its own process with its own space in memory. The browsers should not communicate with one another, nor should they be aware of each other's presence (unless you want them to).
  - Threads are used when a program needs to run a task concurrently. For example, based on the example above, a browser should use threads to create new tabs. In other words, each tab should run as a thread. Since the tabs share the same memory space, a change to the system should reflect all tabs (i.e. Installing *AdBlock* should apply to all tabs/threads).

Question #12

- **Question:** If the parent process terminates, it does not terminate the child process. Is it okay for the process to terminate all threads upon termination. Why or why not?
- **Hint:**
  - Use your answer from Question #10.

Question #12

- **Question:** If the parent process terminates, it does not terminate the child process. Is it okay for the process to terminate all threads upon termination. Why or why not?
- **Hint:**
  - Yes, it is acceptable for the process to terminate all of its threads upon termination. This is because all threads belong to the process, and share the same memory space. If the process terminates, then the threads should terminate as well. It does not make sense, logically or programmatically, for a thread to continue execution if the process has been terminated. Similarly, if *Safari* is terminated, then all tabs that correspond to Safari are also terminated.

Question #13

- **Question:** Assume you are working for Microsoft in the early days (i.e. 1990). Your job is to program MS Word, from scratch. One of your co-workers thinks it is a good idea to implement spell-checking, licence checking, checking for updates, fetching pictures, downloading updates, etc. in a separate process. Explain why or why not this is a good idea? Is there a better alternative to processes, such as threads?
- **Hint:**
  - *No hints. You gotta think for this one!*

Question #13

- **Question: …** Explain why or why not this is a good idea? Is there a better alternative to processes, such as threads?
- **Answer:**
  - This is not a good idea. Performing menial things like spell checking and checking for updates do not require the creation of a process. This wastes CPU time, memory, and other resources. Recall that processes are heavy-weight and threads are light-weight.
  - Activites like spell checking should be done in a thread. This is because the memory space needs to be the same. Spell checking is performed on the current document. It does not make sense to copy the document and spell check the copy, rather than the original document. Creating a copy poses serious programming challenges like conveying the information back to the original process. Hence, it is better to do everything in the same space.
    - In addition, the spell checking thread should be terminate when the process is terminated. It does not make logical sense for a spell-checking process to continue running after the document has been closed. What will it spell check now? Hence, spell checking should be done in a thread.

Any Questions

More Questions

- For the questions on the next slide, refer to the provided C files (inside the *Extra* folder) and answer the following questions:
  - A) How many unique processes are created?
  - B) How many threads are created?
  - C) Draw the corresponding process/thread tree diagram
- Note: No solutions provided. Discuss the answer with your peers.
  - Group study is best study.
  - Although, you can fix the code and then run it.
    - I shall leave that to you.

More Questions

- question13.c
- question14.c
  - *Question: How many times is ‘‘Error:CouldNotJoinThread’’ printed?*
- question15.c
- question16.c
- question17.c
  - *Question: How many times is ‘‘Error:ThreadJoinedException’’ printed?*
  - *Question: How many times is ‘‘Thread XXX Joined Successfully’’ printed?*
  - *Question: How many times is ‘‘DONE!’’ printed?*

Last One

- The following question is to test your understanding of *nix machines and operating systems. This question is not related to concurrency.
  - **Question:** Explain why the *pthread_self()* function never returns an error code/value. Most functions return an error code upon failure or premature termination, but *pthread_self()* does not. Explain why.
- *Note: If you are pressed for time, skip this question.*

GOOD LUCK!