CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 4

The questions here serve the purpose of reviewing concepts from the lecture, and expect the concepts to be tested on the midterm and final. However, they are by no means exhaustive. Anything covered in the lecture and projects can be tested.

1. Definitions and Short Answers - week 4 (10/4 lectures)

1. What is the difference between a **program** and a **process**? Which one is active and which one is passive?
2. What is a **job**?
3. What is a **task**?
4. What is the **text section** of a process's memory?
5. What is the difference between the **data** section and **heap** section of a process's memory?
6. What is kept in the **stack** section of a process's memory?
7. Besides the content of the main memory in a process's address space, what other **state** does a process have (that is maintained by the OS)?
8. What are the **five states** of a process (in the textbook's terminology)?
9. On a single processor, how many processes can be in **running** state at a time? in **ready** state? in **waiting** state?
10. Why would a process enter **waiting** state?
11. From which three states may a process **transition to ready** state?
12. How can a process transition from **running to ready** state without doing any system call?
13. In the process control block (PCB), what is the meaning of the following fields?
    1. **program counter**
    2. **process number**
    3. CPU registers
    4. memory management information
    5. accounting information
    6. I/O status information
14. What is a **thread** of execution?
15. On a single(-core) processor,
    1. how many **processes** can be **running** at a time?
    2. how many **threads** can be **running** at a time?
16. On a multiprocessor system,
    1. how many processes can be running at a time?
    2. how many threads can be running at a time?
17. What is the definition of the **degree of multiprogramming**? Does it reflect
    1. the number of processors in the system?
    2. the number of processes currently running, waiting, ready?
    3. the number of threads currently running, waiting, ready?
    4. the amount of main memory in the system?
    5. the scheduling policy?
18. What is the difference between an **I/O-bound** and a **CPU-bound** process?
19. What does **context switching** mean?
20. What are the possible triggers for context switch?
21. What are the steps taken by the kernel to switch from process P1 to process P2?
22. What kinds of support can hardware provide to help reduce the overhead of context switching?
23. What is a **pid**?
24. In a Unix-like system, how is fork() used for creating a new process?
25. After fork() creates the child process, where in the program does the child process start running?
26. What is the meaning of the return value of fork()?
27. What does exec() do? Does it create a new process? Does exec() return?
28. How does a shell launch a program as a new process using fork() and exec()?
29. What is **copy-on-write**? How does it improve the efficiency of the original implementation of fork()?
30. How many processes are created by the following example? Explain  
    #include <stdio.h>  
    #include <unistd.h>  
    int main() {  
     for (int i=0; i<3; i++) {  
     fork();  
     }  
     return 0;  
     }
31. How does a process terminate voluntarily?
32. What are three reasons a child process may be terminated?
33. What does **cascaded termination** mean?
34. What is an **orphan** process? a **zombie** process?
35. What happens if the parent of a zombie process does not call wait()?
36. What happens on termination of the orphan processes whose parent died before it had a chance to call wait() on its children? What is a solution?
37. What is the name of the "root" process in traditional Unix, with a pid of 1? What is the Linux version called?
38. What is the main advantage of shared memory communication over message passing and why? What must be performed carefully for shared memory to work consistently?
39. What are some ways to ensure shared data are not written simultaneously in inconsistent ways?
40. When is shared memory not an option for two processes to communicate?
41. What does it mean that message passing calls such as send() or receive() may **block**?
42. In the pseudocode for shared memory communication, what are the purposes of variables named in and out? Which variable is modified by consumer and which is modified by producer? Can both the sender and receiver be modifying the same variable at the same time, assuming they can run at the same time?
43. What is "direct communication" (style of message passing) between two processes? How many processes can a link be associated with? How many links can exist between a given pair of processes?
44. In direct communication, how is receive() in asymmetric naming different from receive() in symmetric naming? Can more than one sender process send messages to a given receiver process? If so, can the receiver know the identity of the sender and how?
45. Why does direct communication have the **limited modularity** problem? How is it solved by indirect communication?
46. Does indirect communication allow one sender and multiple receivers to share the same mailbox?
47. Is a **synchronous** call (to a function or procedure) blocking or nonblocking? What about an **asynchronous** call?
48. How does a nonblocking sender/receiver know when the communication is completed (i.e., data has been sent or data is ready to be received)?
49. What is **rendezvous** communication? Are its sender and receiver blocking and nonblocking? How much buffer is required, if any? How do you pronounce "rendezvous"?
50. in **bounded buffer** communication, under what buffer condition does the sender block and does the receiver block?
51. in **infinite buffer** communication, under what buffer condition does the sender block and does the receiver block?
52. what does RPC mean?
53. what is a **stub** function for an RPC on the **client** side, and what does it have to do?
54. What is a **stub** function on the **server** side, and what does it have to do?
55. What do **marshaling** and **demarshaling** mean during an RPC call? Why can't raw data be sent in their original binary representation? What are example types that cause problems?
56. Why do pointers cause problems when passed as parameters or returned in RPC?
57. Can RPC always succeed? What kind of problems can happen?

2. Python Programming

1. Write a *generator* for powers of 2:  
   def gen\_powers\_of\_2():  
    # your code here  
   such that if you test it interactively,  
   >>> g = gen\_powers\_of\_2()  
   >>> next(g)  
   2  
   >>> next(g)  
   4  
   >>> next(g)  
   8  
   >>> [next(g) for i in range(5)]  
   [16, 32, 64, 128, 256]
2. Write a generator for elements of a binary tree in **post-order** (i.e., left child recursively, right child recursively, root). Assume the same kind of tree representation as on slide 11: T = (17, (12, (6, None, None), (14, None, None)), (35, (32, None, None), (40, None, None))) for the tree
3. Rewrite the make\_item() generator on slide 21 to eliminate the for c loop. Hint: slide 13.
4. The consumer() on slide 22 is a "pull"-style communication because the consumer asks for the next item (implicitly called by the for loop) from the generator, which does a yield of the next value. Rewrite the **producer() as a function** and **consumer() as a generator** such that the producer "pushes" the items by calling the send(v) method (slide 23) to send value v to the consumer, which receives the value from yield (as an expression) and calls the use\_item() function on each value. In this case, yield just serves the purpose of "receive".  
   def producer():  
    # basically the same as make\_item() on slide 21, except  
    # 1. instantiate revised consumer as a generator g, and  
    # 2. call next() on g once to kickstart it  
    # 3. inside loop, yield c is replaced by a g.send(v)  
   def consumer():  
    # 1. inifinite loop,  
    # 1.1 c = yield, to receive the value sent by producer,  
    # 1.2 call use\_item(c), found on slide 27

4. Sockets programming in Python

Make a minimal web server at port 8086 in Python by displaying the browser's HTTP request header received and then reply with the HTTP reply header and a minimal page in HTML. Use the template shown on slide 34 and print the request; however, since the request is in bytes type instead of str type, To convert to str for printing, you can call the .decode('utf8') method on the bytes data. The HTTP response should look like this:

HTTP/1.1 200 OK

Date: *fill in your own date time, like unix date command*

Server: Python *or call it anything you like*

Last-Modified: *fill in your own date time*

Content-Length: length of

Content-Type: text/html

Connection: Closed

<html>

<body>

<h1>Hello world</h1>

</body>

</html>

Start your web server from the command line

$ python3 webserv.py

Open your own browser to localhost:8086, and it should render the HTML as a web page.