Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Processes

1. Answer yes/no, and provide a brief explanation.

   (a) Can two processes that are not parent/child be concurrently executing the same program executable?

   (b) Can two running processes share the complete process image in physical memory (not just parts of it)?

   Ans:

   (a) Yes, two processes can run the same program executable, which happens when we run the same executable from the terminal twice.

   (b) No. In general, each process has its own memory image.

2. Consider a process P1 that is executing on a Linux-like OS on a single core system. When P1 is executing, a disk interrupt occurs, causing P1 to go to kernel mode to service that interrupt. The interrupt delivers all the disk blocks that unblock a process P2 (which blocked earlier on the disk read). The interrupt service routine has completed execution fully, and the OS is just about to return back to the user mode of P1. At this point in time, what are the states (ready/running/blocked) of processes P1 and P2?

   (a) State of P1

   (b) State of P2

   Ans:

   (a) P1 is running (b) P2 is ready

3. Consider a process executing on a CPU. Give an example scenario that can cause the process to undergo:

   (a) A voluntary context switch.

   (b) An involuntary context switch.

   Ans:

   (a) A blocking system call.

   (b) Timer interrupt that causes the process to be switched out.

4. Consider a parent process P that has forked a child process C. Now, P terminates while C is still running. Answer yes/no, and provide a brief explanation.

(a) Will C immediately become a zombie?

(b) Will P immediately become a zombie, until reaped by its parent?

Ans:

(a) No, it will be adopted by init.

(b) Yes.

5. A process in user mode cannot execute certain privileged hardware instructions. [T/F]

   Ans: True, some instructions in every CPU's instruction set architecture can only be executed when the CPU is running in a privileged mode (e.g., ring 0 on Intel CPUs).

6. Consider the following CPU instructions found in modern CPU architectures like x86. For each instruction, state if you expect the instruction to be privileged or unprivileged, and justify your answer. For example, if your answer is "privileged", give a one sentence example of what would go wrong if this instruction were to be executable in an unprivileged mode. If your answer is "unprivileged", give a one sentence explanation of why it is safe/necessary to execute the instruction in unprivileged mode.

   (a) Instruction to write into the interrupt descriptor table register.

   (b) Instruction to write into a general purpose CPU register.

   Ans:

   (a) Privileged, because a user process may misuse this ability to redirect interrupts of other processes.

   (b) Unprivileged, because this is a harmless operation that is done often by executing processes.

7. Which of the following C library functions do NOT directly correspond to (similarly named) system calls? That is, the implentations of which of these C library functions are NOT straightforward invocations of the underlying system call?

   (a) `system`, which executes a bash shell command.

   (b) `fork`, which creates a new child process.

   (c) `exit`, which terminates the current process.

   (d) `strlen`, which returns the length of a string.

   Ans: (a), (d)

8. Which of the following actions by a running process will *always* result in a context switch of the running process, even in a non-preemptive kernel design?

   (a) Servicing a disk interrupt, that results in another blocked process being marked as ready/runnable.

   (b) A blocking system call.

   (c) The system call exit, to terminate the current process.

   (d) Servicing a timer interrupt.

Ans: (b), (c)

9. Consider two machines A and B of different architectures, running two different operating systems OS-A and OS-B. Both operating systems are POSIX compliant. The source code of an application that is written to run on machine A must always be rewritten to run on machine B. [T/F]

   Ans: False. If the code is written using the POSIX API, it need not be rewritten for another POSIX compliant system.

10. Consider the scenario of the previous question. An application binary that has been compiled for machine A may have to be recompiled to execute correctly on machine B. [T/F]

    Ans: True. Even if the code is POSIX compliant, the CPU instructions in the compiled executable are different across different CPU architectures.

11. A process makes a system call to read a packet from the network device, and blocks. The scheduler then context-switches this process out. Is this an example of a voluntary context switch or an involuntary context switch?

    Ans: Voluntary context switch.

12. A context switch can occur only after processing a timer interrupt, but not after any other system call or interrupt. [T/F]

    Ans: False, a context switch can also occur after a blocking system call for example.

13. A C program cannot directly invoke the OS system calls and must always use the C library for this purpose. [T/F]

    Ans: False, it is cumbersome but possible to directly invoke system calls from user code.

14. A process undergoes a context switch every time it enters kernel mode from user mode. [T/F]

    Ans: False, after finishing its job in kernel mode, the OS may sometimes decide to go back to the user mode of the same process, without switching to another process.

15. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

    (a) If P does not have any zombie children, then the wait system call returns immediately.

    (b) The wait system call always blocks process P and leads to a context switch.

    (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.

    (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

    Ans: (c)

16. Consider a process P which invokes the default wait system call. For each of the scenarios described below, state the expected behavior of the wait system call, i.e., whether the system call blocks P or if P returns immediately.

    (a) P has no children at all.

(b) P has one child that is still running.

(c) P has one child that has terminated and is a zombie.

(d) P has two children, one of which is running and the other is a terminated zombie.

Ans:

(a) Does not block

(b) Blocks

(c) Does not block

(d) Does not block

17. Consider the wait family of system calls (wait, waitpid etc.) provided by Linux. A parent process uses some variant of the wait system call to wait for a child that it has forked. Which of the following statements is always true when the parent invokes the system call?

    (a) The parent will always block.

    (b) The parent will never block.

    (c) The parent will always block if the child is still running.

    (d) Whether the parent will block or not will depend on the system call variant and the options with which it is invoked.

    Ans: (d)

18. Consider a simple linux shell implementing the command `sleep 100`. Which of the following is an accurate ordered list of system calls invoked by the shell from the time the user enters this command to the time the shell comes back and asks the user for the next input?

    (a) wait-exec-fork

    (b) exec-wait-fork

    (c) fork-exec-wait

    (d) wait-fork-exec

    Ans: (c)

19. Which of the following pieces of information in the PCB of a process are changed when the process invokes the exec system call?

    (a) Process identifier (PID)

    (b) Page table entries

    (c) The value of the program counter stored within the user space context on the kernel stack

    Ans: (a) does not change. (b) and (c) change because the process gets a new memory image (and hence new page table entries pointing to the new image).

20. Which of the following pieces of information about the process are identical for a parent and the newly created child processes, immediately after the completion of the `fork` system call? Answer "identical" or "not identical".

(a) The process identifier.

(b) The contents of the file descriptor table.

Ans: (a) is not identical, as every process has its own unique PID in the system. (b) is identical, as the child gets an exact copy of the parent's file descriptor table.

21. Consider a process P1 that forks P2, P2 forks P3, and P3 forks P4. P1 and P2 continue to execute while P3 terminates. Now, when P4 terminates, which process must wait for and reap P4?

    Ans: init (orphan processes are reaped by init)

22. Consider a process P that executes the fork system call twice. That is, it runs code like this:

    int ret1 = fork(); int ret2 = fork();

    How many direct children of P (i.e., processes whose parent is P) and how many other descendants of P (i.e., processes who are not direct children of P, but whose grandparent or great grandparent or some such ancestor is P) are created by the above lines of code? You may assume that all fork system calls succeed.

    (a) Two direct children of P are created.

    (b) Four direct children of P are created.

    (c) No other descendant of P is created.

    (d) One other descendant of P is created.

    Ans: (a), (d)

23. Consider the x86 instruction"int n" that is executed by the CPU to handle a trap. Which of the following statements is/are true?

    (a) This instruction is always invoked by privileged OS code.

    (b) This instruction causes the CPU to set its EIP to address value n.

    (c) This instruction causes the CPU to lookup the Interrupt Descriptor Table (IDT) using the value n as an index.

    (d) This instruction is always executed by the CPU only in response to interrupts from external hardware, and never due to any code executed by the user.

    Ans: (c)

24. Consider the following scheduling policy implemented by an OS, in which a user can set numerical priorities for processes running in the system. The OS scheduler maintains all ready processes in a strict priority queue. When the CPU is free, it extracts the ready process with the highest priority (breaking ties arbitrarily), and runs it until the process blocks or terminates. Which of the following statements is/are true about this scheduling policy?

    (a) This scheduler is an example of a non-preemptive scheduling policy.

    (b) This scheduling policy can result in the starvation of low priority processes.

    (c) This scheduling policy guarantees fairness across all active processes.

    (d) This scheduling policy guarantees lowest average turnaround time for all processes.

Ans: (a), (b)

25. Consider the following scheduling policy implemented by an OS. Every time a process is scheduled, the OS runs the process for a maximum of 10 milliseconds or until the process blocks or terminates itself before 10 milliseconds. Subsequently, the OS moves on to the next ready process in the list of processes in a round-robin fashion. Which of the following statements is/are true about this scheduling policy?

    (a) This policy cannot be efficiently implemented without hardware support for timer interrupts.

    (b) This scheduler is an example of a non-preemptive scheduling policy.

    (c) This scheduling policy can sometimes result in involuntary context switches.

    (d) This scheduling policy prioritizes processes with shorter CPU burst times over processes that run for long durations.

    Ans: (a), (c)

26. Consider a process P that needs to save its CPU execution context (values of some CPU registers) on some stack when it makes a function call or system call. Which of the following statements is/are true?

    (a) During a system call, when transitioning from user mode to kernel mode, the context of the process is saved on its kernel stack.

    (b) During a function call in user mode, the context of the process is saved on its user stack.

    (c) During a function call in kernel mode, the context of the process is saved on its user stack.

    (d) During a function call in kernel mode, the context of the process is saved on its kernel stack.

    Ans: (a), (b), (d)

27. For each of the events below, state whether the execution context of a running process P will be saved on the user stack of P or on the kernel stack.

    (a) P makes a function call in user mode. **Ans:** user stack

    (b) P makes a function call in kernel mode. **Ans:** kernel stack

    (c) P makes a system call and moves from user mode to kernel mode. **Ans:** kernel stack

    (d) P is switched out by the CPU scheduler. **Ans:** kernel stack

28. Which of the following statements is/are true regarding how the trap instruction (e.g., int n in x86) is invoked when a trap occurs in a system?

    (a) When a user makes a system call, the trap instruction is invoked by the kernel code handling the system call

    (b) When a user makes a system call, the trap instruction is invoked by userspace code (e.g., user program or a library)

    (c) When an external I/O device raises an interrupt, the trap instruction is invoked by the device driver handling the interrupt

    (d) When an external I/O device raises an interrupt signaling the completion of an I/O request, the trap instruction is invoked by the user process that raised the I/O request

Ans: (b)

29. Which of the following statements is/are true about a context switch?

    (a) A context switch from one process to another will happen every time a process moves from user mode to kernel mode

    (b) For preemptive schedulers, a trap of any kind always leads to a context switch

    (c) A context switch will always occur when a process has made a blocking system call, irrespective of whether the scheduler is preemptive or not

    (d) For non-preemptive schedulers, a process that is ready/willing to run will not be context switched out

    Ans: (c), (d)

30. When a process makes a system call and runs kernel code:

    (a) How does the process obtain the address of the kernel instruction to jump to?

    (b) Where is the userspace context of the process (program counter and other registers) stored during the transition from user mode to kernel mode?

    Ans:

    (a) From IDT (interrupt descriptor table) (b) on kernel stack of process (which is linked from the PCB)

31. Which of the following operations by a process will definitely cause the process to move from user mode to kernel mode? Answer yes (if a change in mode happens) or no.

    (a) A process invokes a function in a userspace library.
    Ans: no

    (b) A process invokes the `kill` system call to send a signal to another process.
    Ans: yes

32. Consider the following events that happen during a context switch from (user mode of) process P to (user mode of) process Q, triggered by a timer interrupt that occurred when P was executing, in a Unix-like operating system design studied in class. Arrange the events in chronological order, starting from the earliest to the latest.

    (A) The CPU program counter moves from the kernel address space of P to the kernel address space of Q.

    (B) The CPU executing process P moves from user mode to kernel mode.

    (C) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.

    (D) The CPU program counter moves from the kernel address space of Q to the user address space of Q.

    (E) The OS scheduler code is invoked.

    Ans:

    B E C A D

33. Consider a system with two processes P and Q, running a Unix-like operating system as studied in class. Consider the following events that may happen when the OS is concurrently executing P and Q, while also handling interrupts.

(A) The CPU program counter moves from pointing to kernel code in the kernel mode of process P to kernel code in the kernel mode of process Q.

(B) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.

(C) The CPU executing process P moves from user mode of P to kernel mode of P.

(D) The CPU executing process P moves from kernel mode of P to user mode of P.

(E) The CPU executing process Q moves from the kernel mode of Q to the user mode of Q.

(F) The interrupt handling code of the OS is invoked.

(G) The OS scheduler code is invoked.

For each of the two scenarios below, list out the chronological order in which the events above occur. Note that all events need not occur in each question.

(a) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to return to process P.

(b) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to context switch to process Q, and the system ends up in the user mode of Q.

Ans:

(a) C F G D

(b) C F G B A E

34. Consider the following three processes that arrive in a system at the specified times, along with the duration of their CPU bursts. Process P1 arrives at time t=0, and has a CPU burst of 10 time units. P2 arrives at t=2, and has a CPU burst of 2 units. P3 arrives at t=3, and has a CPU burst of 3 units. Assume that the processes execute only once for the duration of their CPU burst, and terminate immediately. Calculate the time of completion of the three processes under each of the following scheduling policies. For each policy, you must state the completion time of all three processes, P1, P2, and P3. Assume there are no other processes in the scheduler's queue. For the preemptive policies, assume that a running process can be immediately preempted as soon as the new process arrives (if the policy should decide to premempt).

(a) First Come First Serve

(b) Shortest Job First (non-preemptive)

(c) Shortest Remaining Time First (preemptive)

(d) Round robin (preemptive) with a time slice of (atmost) 5 units per process

Ans:

(a) FCFS: P1 at 10, P2 at 12, P3 at 15

(b) SJF: same as above

(c) SRTF: P2 at 4, P3 at 7, P1 at 15

(d) RR: P2 at 7, P3 at 10, P1 at 15

35. Consider a system with a single CPU core and three processes A, B, C. Process A arrives at $t = 0$, and runs on the CPU for 10 time units before it finishes. Process B arrives at $t = 6$, and requires an initial CPU time of 3 units, after which it blocks to perform I/O for 3 time units. After returning from I/O wait, it executes for a further 5 units before terminating. Process C arrives at $t = 8$, and runs for 2 units of time on the CPU before terminating. For each of the scheduling policies below, calculate the time of completion of each of the three processes. Recall that only the size of the current CPU burst (excluding the time spent for waiting on I/O) is considered as the "job size" in these schedulers.

    (a) First Come First Serve (non-preemptive).

    Ans: A=10, B=21, C=15. A finishes at 10 units. First run of B finishes at 13. C completes at 15. B restarts at 16 and finishes at 21.

    (b) Shortest Job First (non-preemptive)

    Ans: A=10, B=23, C=12. A finishes at 10 units. Note that the arrival of shorter jobs B and C does not preempt A. Next, C finishes at 12. First task of B finishes at 15, B blocks from 15 to 18, and finally completes at 23 units.

    (c) Shortest Remaining Time First (preemptive)

    Ans: A=15, B=20, C=11. A runs until 6 units. Then the first task of B runs until 9 units. Note that the arrival of C does not preempt B because it has a shorter remaining time. C completes at 11. B is not ready yet, so A runs for another 4 units and completes at 15. Note that the completion of B's I/O does not preempt A because A's remaining time is shorter. B finally restarts at 15 and completes at 20.

36. Consider the various CPU scheduling mechanisms and techniques used in modern schedulers.

    (a) Describe one technique by which a scheduler can give higher priority to I/O-bound processes with short CPU bursts over CPU-bound processes with longer CPU bursts, without the user explicitly having to specify the priorities or CPU burst durations to the scheduler.

    Ans: Reducing priority of a process that uses up its time slice fully (indicating it is CPU bound)

    (b) Describe one technique by which a scheduler can ensure that high priority processes do not starve low priority processes indefinitely.

    Ans: Resetting priority of processes periodically, or round robin.

37. Consider the following C program. Assume there are no syntax errors and the program executes correctly. Assume the fork system calls succeed. What is the output printed to the screen when we execute the below program?

```
void main(argc, argv) {

  for(int i = 0; i < 4; i++) {
    int ret = fork();
    if(ret == 0)
      printf("child %d\n", i);
  }
}
```

Ans: The statement "child i" is printed 2^i times for i=0 to 3

38. Consider a parent process P that has forked a child process C in the program below.

```
int a = 5;
int fd = open(...) //opening a file
int ret = fork();
if(ret >0) {
  close(fd);
  a = 6;
  ...
}
else if(ret==0) {
  printf("a=%d\n", a);
  read(fd, something);
}
```

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

(a) What is the value of the variable `a` as printed in the child process, when it is scheduled next? Explain.

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

Ans:

(a) 5. The value is only changed in the parent.

(b) Yes, the file is only closed in the parent.

39. Consider the following pseudocode. Assume all system calls succeed and there are no other errors in the code.

```
int ret1 = fork(); //fork1
int ret2 = fork(); //fork2
int ret3 = fork(); //fork3
wait();
wait();
wait();
```

Let us call the original parent process in this program as P. Draw/describe a family tree of P and all its descendents (children, grand children, and so on) that are spawned during the execution of this program. Your tree should be rooted at P. Show the spawned descendents as nodes in the tree, and connect processes related by the parent-child relationship with an arrow from parent to child. Give names containing a number for descendents, where child processes created by fork "i" above should have numbers like "i1", "i2", and so on. For example, child processes created by fork3 above should have names C31, C32, and so on.

Ans: P has three children, one in each fork statement: C11, C21, C31. C11 has two children in the second and third fork statements: C22, C32. C21 and C22 also have a child each in the third fork statement: C33 and C34.

40. Consider a parent process that has forked a child in the code snippet below.

```
int count = 0;
ret = fork();
if(ret == 0)  {
printf("count in child=%d\n", count);
}
else  {
count = 1;
}
```

The parent executes the statement "count = 1" before the child executes for the first time. Now, what is the value of count printed by the code above?

Ans: 0 (the child has its own copy of the variable)

41. Consider the following sample code from a simple shell program.

```
command = read_from_user();
int rc = fork();
if(rc == 0) { //child
  exec(command);
}
else {//parent
  wait();
}
```

Now, suppose the shell wishes the redirect the output of the command not to STDOUT but to a file "foo.txt". Show how you would modify the above code to achieve this output redirection. You can indicate your changes next to the code above.

Ans:

Modify the child code as follows.

```
close(STDOUT_FILENO)
open("foo.txt")
exec(command)
```

42. What is the output of the following code snippet? You are given that the `exec` system call in the child does not succeed.

```
int ret = fork();
if(ret==0) {
  exec(some_binary_that_does_not_exec);
  printf(``child\n'');
  }
else {
  wait();
  printf(``parent\n'');
  }
```

Ans:

```
child
parent
```

43. Consider the following code snippet, where a parent process forks a child process. The child performs one task during its lifetime, while the parent performs two different tasks.

```
int ret = fork();
if(ret == 0) { do_child_task(); }
else { do_parent_task1();
       do_parent_task2(); }
```

With the way the code is written right now, the user has no control over the order in which the parent and child tasks execute, because the scheduling of the processes is done by the OS. Below are given two possible orderings of the tasks that the user wishes to enforce. For each part, briefly describe how you will modify the code given above to ensure the required ordering of tasks. You may write your answer in English or using pseudocode.

Note that you cannot change the OS scheduling mechanism in any way to solve this question. If a process is scheduled by the OS before you want its task to execute, you must use mechanisms like system calls and IPC techniques available to you in userspace to delay the execution of the task till a suitable time.

(a) We want the parent to start execution of both its tasks only after the child process has finished its task and has terminated.

Ans: Parent does wait() until child finishes, and then starts its tasks.

(b) We want the child process to execute its task after the parent process has finished its first task, but before it runs its second task. The parent must not execute its second task until the child has completed its task and has terminated.

Ans: Many solutions are possible. Parent and child share two pipes (or a socket). Parent writes to one pipe after completing task 1 and child blocks on this pipe read before starting its task. Child writes to pipe 2 after finishing its task, and parent blocks on this pipe read before starting its second task. (Or parent can use wait to block for child termination, like in previous part.)

44. What are the possible outputs printed from this program shown below? You may assume that the program runs on a modern Linux-like OS. You may ignore any output generated from "some_executable". You must consider all possible scenarios of the system calls succeeding as well as failing. In your answer, clearly list down all the possible scenarios, and the output of the program in each of these scenarios.

```
int ret = fork();
if(ret == 0) {
  printf(``Hello1\n'');
  exec(``some_executable'');
  printf(``Hello2\n'');
} else if(ret > 0) {
  wait();
  printf(``Hello3\n'');
} else {
  printf(``Hello4\n'');
}
```

Ans: Case I: fork and exec succeed. Hello1, Hello3 are printed. Case II: fork succeeds but exec fails. Hello1, Hello2, Hello3 are printed. Case III: fork fails. Hello4 is printed.

45. Consider the following sample code from a simple shell program.

```
int rc1 = fork();
if(rc1 == 0) {
  exec(cmd1);
}
else {
  int rc2 = fork();
  if(rc2 == 0) {
  exec(cmd2);
  }
  else {
    wait();
    wait();
  }
}
```

(a) In the code shown above, do the two commands `cmd1` and `cmd2` execute serially (one after the other) or in parallel? **Ans:** parallel.

(b) Indicate how you would modify the code above to change the mode of execution from serial to parallel or vice versa. That is, if you answered "serial" in part (a), then you must change the code to execute the commands in parallel, and vice versa. Indicate your changes next to the code snippet above. **Ans:** move the first wait to before second fork.

46. Consider the following code snippet running on a modern Linux operating systems (with a reasonable preemptive scheduling policy as studied in class). Assume that there are no other interfering processes in the system. Note that the executable "good_long_executable" runs for 100 seconds, prints the line "Hello from good executable" to screen, and terminates. On the other hand, the file "bad_executable" does not exist and will cause the `exec` system call to fail.

```
int ret1 = fork();
if(ret1 == 0) { //Child 1
  printf("Child 1 started\n");
  exec("good_long_executable");
  printf("Child 1 finished\n");
}
else { //Parent
  int ret2 == fork();
  if(ret2 == 0) { //Child 2
    sleep(10); //Sleeping allows child 1 to begin execution
    printf("Child 2 started\n");
    exec("bad_executable");
    printf("Child 2 finished\n");
  } //end of Child 2
  else { //Parent
    wait();
    printf("Child reaped\n");
    wait();
    printf("Parent finished\n");
  }
}
```

Write down the output of the above program.

Ans:

Child 1 started
Child 2 started
(Some error message from the wrong executable)
Child 2 finished
Child reaped
Hello from good executable
Parent finished

47. What is the output printed by the following snippet of pseudocode? If you think there is more than one possible answer depending on the execution order of the processes, then you must list all possible outputs.

```
int fd[2];
pipe(fd);
int rc = fork();
if(rc == 0) { //child
  close(fd[1]);
  printf(``child1\n'');
  read(fd[0], bufc, bufc_size);
  printf(``child2\n'');
}
else {//parent
  close(fd[0]);
  printf(``parent1\n'');
  write(fd[1], bufp, bufp_size);
  wait();
  printf(``parent2\n'');
}
```

Ans: If child scheduled before parent: child1, parent1, child2, parent 2. If parent scheduled before child, parent1, child1, child2, parent2.

48. What is the output of the following program? Only relevant code snippets are shown, and you may assume that the code compiles and executes successfully.

```
int a = 2;
while(a > 0) {
    int ret = fork();
    if(ret == 0) {
        a++;
        printf("a=%d\n", a);
    }
    else {
        wait(NULL);
        a--;
    }
}
```

Ans: The code goes into an infinite while+fork loop, as every child that is forked will execute the same code with a higher value of "a" and hence the while loop never terminates. There will be an infinite number of child processes that will print a=3,4,5, and so on, until either fork fails or the system exchausts some other resource.

49. What is the output of the following program? Only relevant code snippets are shown, and you may assume that the code compiles and executes successfully.

```
int a = 2;
while(a > 0) {
    int ret = fork();
    if(ret == 0) {
        a++;
        printf("a=%d\n", a);
        execl("/bin/ls", "/bin/ls", NULL);
    }
    else {
        wait(NULL);
        a--;
    }
}
```

Ans:

```
a=3
<output of ls>
a=2
<output of ls>
```

50. Consider an application that is composed of one master process and multiple worker processes that are forked off the master at the start of application execution. All processes have access to a pool of shared memory pages, and have permissions to read and write from it. This shared memory region (also called the request buffer) is used as follows: the master process receives incoming requests from clients over the network, and writes the requests into the shared request buffer. The worker processes must read the request from the request buffer, process it, and write the response back into the same region of the buffer. Once the response has been generated, the server must reply back to the client. The server and worker processes are single-threaded, and the server uses event-driven I/O to communicate over the network with the clients (you must not make these processes multi threaded). You may assume that the request and the response are of the same size, and multiple such requests or responses can be accommodated in the request buffer. You may also assume that processing every request takes similar amount of CPU time at the worker threads.

    Using this design idea as a starting point, describe the communication and synchronization mechanisms that must be used between the server and worker processes, in order to let the server correctly delegate requests and obtain responses from the worker processes. Your design must ensure that every request placed in the request buffer is processed by one and only one worker thread. You must also ensure that the system is efficient (e.g., no request should be kept waiting if some worker is free) and fair (e.g., all workers share the load almost equally). While you can use any IPC mechanism of your choice, ensure that your system design is practical enough to be implementable in a modern multicore system running an OS like Linux. You need not write any code, and a clear, concise and precise description in English should suffice.

Ans: Several possible solutions exist. The main thing to keep in mind is that the server should be able to assign a certain request in the buffer to a worker, and the worker must be able to notify completion. For example, the master can use pipes or sockets or message queues with each worker. When it places a request in the shared memory, it can send the position of the request to one of the workers. Workers listen for this signal from the master, process the request, write the response, and send a message back to the master that it is done. The master monitors the pipes/sockets of all workers, and assigns the next request once the previous one is done.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Memory

1. Provide one advantage of using the slab allocator in Linux to allocate kernel objects, instead of simply allocating them from a dynamic memory heap.

   Ans: A slab allocator is fast because memory is preallocated. Further, it avoids fragmentation of kernel memory.

2. In a 32-bit architecture machine running Linux, for every physical memory address in RAM, there are at least 2 virtual addresses pointing to it. That is, every physical address is mapped at least twice into the virtual address space of some set of processes. [T/F]

   Ans: F (this may be true of simple OS like xv6 studied in class, but not generally true)

3. Consider a system with N bytes of physical RAM, and M bytes of virtual address space per process. Pages and frames are K bytes in size. Every page table entry is P bytes in size, accounting for the extra flags required and such. Calculate the size of the page table of a process.

   Ans: M/K * P

4. The memory addresses generated by the CPU when executing instructions of a process are called logical addresses. [T/F]

   Ans: T

5. When a C++ executable is run on a Linux machine, the kernel code is part of the executable generated during the compilation process. [T/F]

   Ans: F (it is only part of the virtual address space)

6. When a C++ executable is run on a Linux machine, the kernel code is part of the virtual address space of the running process. [T/F]

   Ans: T

7. Consider a Linux-like OS running on x86 Intel CPUs. Which of the following events requires the OS to update the page table pointer in the MMU (and flush the changes to the TLB)? Answer "update" or "no update".

   (a) A process moves from user mode to kernel mode.

   Ans: no update

   (b) The OS switches context from one process to another.

   Ans: update

8. Consider a process that has just forked a child. The OS implements a copy-on-write fork. At the end of the fork system call, the OS does not perform a context switch and will return back to the user mode of the parent process. Now, which of the following entities are updated at the end of a successful implementation of the fork system call? Answer "update" or "no update".

   (a) The page table of the parent process.
   Ans: update because the parent's pages must be marked read-only.

   (b) The page table information in the MMU and the TLB.
   Ans: update because the parent's pages must be marked read-only.

9. A certain page table entry in the page table of a process has both the valid and present bits set. Describe what happens on a memory access to a virtual address belonging to this page table entry.

   (a) What happens at the TLB? (hit/miss/cannot say)
   Ans: cannot say

   (b) Will a page fault occur? (yes/no/cannot say)
   Ans no

10. A certain page table entry in the page table of a process has the valid bit set but the present bit unset. Describe what happens on a memory access to a virtual address belonging to this page table entry.

    (a) What happens at the TLB? (hit/miss/cannot say)
    Ans: miss

    (b) Will a page fault occur? (yes/no/cannot say)
    Ans yes

11. Consider the page table entries within the page table of a process that map to kernel code/data stored in RAM, in a Linux-like OS studied in class.

    (a) Are the physical addresses of the kernel code/data stored in the page tables of various process always the same? (yes/no/cannot say)
    Ans: yes, because there is only one copy of kernel code in RAM

    (b) Does the page table of every process have page table entries pointing to the kernel code/data? (yes/no/cannot say)
    Ans: yes, because every process needs to run kernel code in kernel mode

12. Consider a process P running in a Linux-like operating system that implements demand paging. The page/frame size in the system is 4KB. The process has 4 pages in its heap. The process stores an array of 4K integers (size of integer is 4 bytes) in these 4 pages. The process then proceeds to access the integers in the array sequentially. Assume that none of these 4 pages of the heap are initially in physical memory. The memory allocation policy of the OS allocates only 3 physical frames at any point of time, to the store these 4 pages of the heap. In case of a page fault and all 3 frames have been allocated to the heap of the process, the OS uses a LRU policy to evict one of these pages to make space for the new page. Approximately what percentage of the 4K accesses to array elements will result in a page fault?

(a) Almost 100%

(b) Approximately 25%

(c) Approximately 75%

(d) Approximately 0.1%

Ans: (d)

13. Given below are descriptions of different entries in the page table of a process, with respect to which bits are set and which are not set. Accessing which of the page table entries below will always result in the MMU generating a trap to the OS during address translation?

    (a) Page with both valid and present bits set

    (b) Page with valid bit set but present bit unset

    (c) Page with valid bit unset

    (d) Page with valid, present, and dirty bits set

    Ans: (b), (c)

14. Which of the following statements is/are true regarding the memory image of a process?

    (a) Memory for non-static local variables of a function is allocated on the heap dynamically at run time

    (b) Memory for arguments to a function is allocated on the stack dynamically at run time

    (c) Memory for static and global variables is allocated on the stack dynamically at run time

    (d) Memory for the argc, argv arguments to the main function is allocated in the code/data section of the executable at compile time

    Ans: (b)

15. Consider a process P in a Linux-like operating system that implements demand paging. Which of the following pages in the page table of the process will have the valid bit set but the present bit unset?

    (a) Pages that have been used in the past by the process, but were evicted to swap space by the OS due to memory pressure

    (b) Pages that have been requested by the user using mmap/brk/sbrk system calls, but have not yet been accessed by the user, and hence not allocated physical memory frames by the OS

    (c) Pages corresponding to unused virtual addresses in the virtual address space of the process

    (d) Pages with high virtual addresses mapping to OS code and data

    Ans: (a), (b)

16. Consider a process P in a Linux-like operating system that implements demand paging. For a particular page in the page table of this process, the valid and present bits are both set. Which of the following are possible outcomes that can happen when the CPU accesses a virtual address in this page of the process? Select all outcomes that are possible.

(a) TLB hit (virtual address found in TLB)

(b) TLB miss (virtual address not found in TLB)

(c) MMU walks the page table (to translate the address)

(d) MMU traps to the OS (due to illegal access)

Ans: (a), (b), (c), (d)

17. Consider a process P in a Linux-like operating system that implements demand paging using the LRU page replacement policy. You are told that the i-th page in the page table of the process has the accessed bit set. Which of the following statements is/are true?

(a) This bit was set by OS when it allocated a physical memory frame to the page

(b) This bit was set by MMU when the page was accessed in the recent past

(c) This page is likely to be evicted by the OS page replacement policy in the near future

(d) This page will always stay in physical memory as long as the process is alive

Ans: (b)

18. Which of the following statements is/are true regarding the functions of the OS and MMU in a modern computer system?

(a) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is created in the system

(b) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is context switched in by the CPU scheduler

(c) MMU traps to OS every time an address is not found in the TLB cache

(d) MMU traps to OS every time it cannot translate an address using the page table available to it

Ans: (b), (d)

19. Consider a modern computer system using virtual addressing and translation via MMU. Which of the following statements is/are valid advantages of using virtual addressing as opposed to directly using physical addresses to fetch instructions and data from main memory?

(a) One does not need to know the actual addresses of instructions and data in main memory when generating compiled executables.

(b) One can easily provide isolation across processes by limiting the physical memory that is mapped into the virtual address space of a process.

(c) Using virtual addressing allows us to hide the fact that user's memory is allocated non-contiguously, and helps provide a simplified view to the user.

(d) Memory access using virtual addressing is faster than directly accessing memory using physical addresses.

Ans: (a), (b), (c)

20. Consider a process running on a system with a 52-bit CPU (i.e., virtual addresses are 52 bits in size). The system has a physical memory of 8GB. The page size in the system is 4KB, and the size of a page table entry is 4 bytes. The OS uses hierarchical paging. Which of the following statements is/are true? You can assume 2^{10} = 1K, 2^{20} = 1M, and so on.

    (a) We require a 4-level page table to keep track of the virtual address space of a process.

    (b) We require a 5-level page table to keep track of the virtual address space of a process.

    (c) The most significant 9 bits are used to index into the outermost page directory by the MMU during address translation.

    (d) The most significant 40 bits of a virtual address denote the page number, and the least significant 12 bits denote the offset within a page.

    Ans: (a), (d)

21. Consider the following line of code in a function of a process.
    ```
    int *x = (int *)malloc(10 * sizeof(int));
    ```
    When this function is invoked and executed:

    (a) Where is the memory for the variable `x` allocated within the memory image of the process? (stack/heap)

    Ans: stack

    (b) Where is the memory for the 10 integer variables allocated within the memory image of the process? (stack/heap)

    Ans: heap

22. Consider an OS that is not using a copy-on-write implementation for the `fork` system call. A process P has spawned a child C. Consider a virtual address v that is translated to physical address $A_p(v)$ using the page table of P, and to $A_c(v)$ using the page table of C.

    (a) For which virtual addresses v does the relationship $A_p(v) = A_c(v)$ hold?

    Ans: For kernel space addresses, shared libraries and such.

    (b) For which virtual addresses v does the relationship $A_p(v) = A_c(v)$ not hold?

    Ans: For userspace part of memory image, e.g., code, data, stack, heap.

23. Consider a system with paging-based memory management, whose architecture allows for a 4GB virtual address space for processes. The size of logical pages and physical frames is 4KB. The system has 8GB of physical RAM. The system allows a maximum of 1K (=1024) processes to run concurrently. Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of *all* processes in the system. Assume that each page table entry requires an additional 10 bits (beyond the frame number) to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations.

    Ans:

    Number of physical frames = $2^{33}/2^{12} = 2^{21}$. Each PTE has frame number (21 bits) and flags (10 bits) $\approx$ 4 bytes. The total number of pages per process is $2^{32}/2^{12}$=2^{20}, so total size of inner page table pages is $2^{20} \times 4$ = 4MB.

Each page can hold $2^{12}/4 = 2^{10}$ PTEs, so we need $2^{20}/2^{10}$ PTEs to point to inner page tables, which will fit in a single outer page table. So the total size of page tables of one process is 4MB + 4KB. For 1K process, the total memory consumed by page tables is 4GB + 4MB.

24. Consider a simple system running a single process. The size of physical frames and logical pages is 16 bytes. The RAM can hold 3 physical frames. The virtual addresses of the process are 6 bits in size. The program generates the following 20 virtual address references as it runs on the CPU: `0, 1, 20, 2, 20, 21, 32, 31, 0, 60, 0, 0, 16, 1, 17, 18, 32, 31, 0, 61`. (Note: the 6-bit addresses are shown in decimal here.) Assume that the physical frames in RAM are initially empty and do not map to any logical page.

    (a) Translate the virtual addresses above to logical page numbers referenced by the process. That is, write down the reference string of 20 page numbers corresponding to the virtual address accesses above. Assume pages are numbered starting from 0, 1, ...

    (b) Calculate the number of page faults genrated by the accesses above, assuming a FIFO page replacement algorithm. You must also correctly point out which page accesses in the reference string shown by you in part (a) are responsible for the page faults.

    (c) Repeat (b) above for the LRU page replacement algorithm.

    (d) What would be the lowest number of page faults achievable in this example, assuming an optimal page replacement algorithm were to be used? Repeat (b) above for the optimal algorithm.

    Ans:

    (a) For 6 bit virtual addresses, and 4 bit page offsets (page size 16 bytes), the most significant 2 bits of a virtual address will represent the page number. So the reference string is 0, 0, 1, 0, 1, 1, 2, 1, 0, 3 (repeated again).

    (b) Page faults with FIFO = 8. Page faults on 0,1,2,3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.

    (c) Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3.

    (d) The optimum algorithm will replace the page least likely to be used in future, and would look like LRU above.

25. Consider a system with only virtual addresses, but no concept of virtual memory or demand paging. Define *total memory access time* as the time to access code/data from an address in physical memory, including the time to resolve the address (via the TLB or page tables) and the actual physical memory access itself. When a virtual address is resolved by the TLB, experiments on a machine have empirically observed the total memory access time to be (an approximately constant value of) t_h. Similarly, when the virtual address is not in the TLB, the total memory access time is observed to be t_m. If the average total memory access time of the system (averaged across all memory accesses, including TLB hits as well as misses) is observed to be t_x, calculate what fraction of memory addresses are resolved by the TLB. In other words, derive an expression for the TLB hit rate in terms of t_h, t_m, and t_x. You may assume $t_m > t_h$.

    Ans: We have $t_x = h * t_h + (1 - h) * t_m$, so $t_h = \frac{t_m - t_x}{t_m - t_h}$

26. 4. Consider a system with a 6 bit virtual address space, and 16 byte pages/frames. The mapping from virtual page numbers to physical frame numbers of a process is (0,8), (1,3), (2,11), and (3,1). Translate the following virtual addresses to physical addresses. Note that all addresses are in decimal. You may write your answer in decimal or binary.

    (a) 20

    (b) 40

    Ans:

    (a) 20 = 01 0100 = 11 0100 = 52

    (b) 40 = 10 1000 = 1011 1000 = 184

27. Consider a system with several running processes. The system is running a modern OS that uses virtual addresses and demand paging. It has been empirically observed that the memory access times in the system under various conditions are: t1 when the logical memory address is found in TLB cache, t2 when the address is not in TLB but does not cause a page fault, and t3 when the address results in a page fault. This memory access time includes all overheads like page fault servicing and logical-to-physical address translation. It has been observed that, on an average, 10% of the logical address accesses result in a page fault. Further, of the remaining virtual address accesses, two-thirds of them can be translated using the TLB cache, while one-third require walking the page tables. Using the information provided above, calculate the average expected memory access time in the system in terms of t1,t2, and t3.

    Ans: 0.6*t1 + 0.3*t2 + 0.1*t3

28. Consider a system where each process has a virtual address space of 2^v bytes. The physical address space of the system is 2^p bytes, and the page size is 2^k bytes. The size of each page table entry is 2^e bytes. The system uses hierarchical paging with l levels of page tables, where the page table entries in the last level point to the actual physical pages of the process. Assume $l \geq 2$. Let v_0 denote the number of (most significant) bits of the virtual address that are used as an index into the outermost page table during address translation.

    (a) What is the number of logical pages of a process?

    (b) What is the number of physical frames in the system?

    (c) What is the number of PTEs that can be stored in a page?

    (d) How many pages are required to store the innermost PTEs?

    (e) Derive an expression for l in terms of v, p, k, and e.

    (f) Derive an expression for v_0 in terms of l, v, p, k, and e.

    Ans:

    (a) 2^{v-k}

    (b) 2^{p-k}

    (c) 2^{k-e}

    (d) 2^{v-k} / 2^{k-e} = 2^{v+e-2k}

(e) The least significant k of v bits indicate offset within a page. Of the remaining $v - k$ bits, $k - e$ bits will be used to index into the page tables at every level, so the number of levels l = ceil $\frac{v-k}{k-e}$

(f) $v - k - (l - 1) * (k - e)$

29. Consider an operating system that uses 48-bit virtual addresses and 16KB pages. The system uses a hierarchical page table design to store all the page table entries of a process, and each page table entry is 4 bytes in size. What is the total number of pages that are required to store the page table entries of a process, across all levels of the hierarchical page table?

Ans: Page size = 2^{14} bytes. So, the number of page table entries = $2^{48}/2^{14} = 2^{34}$. Each page can store 16KB/4 = 2^{12} page table entries. So, the number of innermost pages = 2^{34} / $2^{12} = 2^{22}$.

Now, pointers to all these innermost pages must be stored in the next level of the page table, so the next level of the page table has 2^{22} / $2^{12} = 2^{10}$ pages. Finally, a single page can store all the 2^{10} page table entries, so the outermost level has one page.

So, the total number of pages that store page table entries is $2^{22} + 2^{10} + 1$.

30. Consider a memory allocator that uses the buddy allocation algorithm to satisfy memory requests. The allocator starts with a heap of size 4KB (4096 bytes). The following requests are made to the allocator by the user program (all sizes requested are in bytes): ptr1 = malloc(500); ptr2 = malloc(200); ptr3 = malloc(800); ptr4 = malloc(1500). Assume that the header added by the allocator is less than 10 bytes in size. You can make any assumption about the implementation of the buddy allocation algorithm that is consistent with the description in class.

(a) Draw a figure showing the status of the heap after these 4 allocations complete. Your figure must show which portions of the heap are assigned and which are free, including the sizes of the various allocated and free blocks.

(b) Now, suppose the user program frees up memory allocations of ptr2, ptr3, and ptr4. Draw a figure showing the status of the heap once again, after the memory is freed up and the allocation algorithm has had a chance to do any possible coalescing.

Ans:

(a) [512 B][256 B] 256 B free [1024 B][2048 B]

(b) [512 B] 512 B free, 1024 B free, 2048 B free. No further coalescing is possible.

31. Consider a system with 8-bit virtual and physical addresses, and 16 byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical pages in the following manner: logical page 0 maps to physical page 6, 1 maps to 3, 2 maps to 11, and logical page 5 is not mapped to any physical page yet. All the other pages in the virtual address space of the process are marked invalid in the page table. The MMU is given a pointer to this page table for address translation. Further, the MMU has a small TLB cache that stores two entries, for logical pages 0 and 2. For each virtual address shown below, describe what happens when that address is accessed by the CPU. Specifically, you must answer what happens at the TLB (hit or miss?), MMU (which page table entry is accessed?), OS (is there a trap of any kind?), and the physical memory (which physical address is accessed?). You may write the translated physical address in binary format. (Note that it is not implied that the accesses below happen one after the other; you must solve each part of the question independently using the information provided above.)

(a) Virtual address 7

(b) Virtual address 20

(c) Virtual address 70

(d) Virtual address 80

Ans:

(a) 7 = 0000 (page number) + 0111 (offset) = logical page 0. TLB hit. No page table walk. No OS trap. Physical address 0110 0111 is accessed.

(b) 20 = 0001 0100 = logical page 1. TLB miss. MMU walks page table. Physical address 0011 0100

(c) 70 = 0100 0110 = logical page 4. TLB miss. MMU accesses page table and discovers it is an invalid entry. MMU raises trap to OS.

(d) 80 = 0101 0000 = logical page 5. TLB miss. MMU accesses page table and discovers page not present. MMU raises a page fault to the OS.

32. Consider a system with 8-bit addresses and 16-byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical frames in the following manner: logical page 0 maps to physical frame 2, page 1 maps to frame 0, page 2 maps to frame 1, and page 3 is not mapped to any physical frame. The process may not use more than 3 physical frames. On a page fault, the demand paging system uses the LRU policy to evict a page. The MMU has a TLB cache that can store 2 entries. The TLB cache also uses the LRU policy to store the most recently used mappings in cache. Now, the process accesses the following logical addresses in order: 7, 17, 37, 20, 40, 60.

(a) Out of the 6 memory accesses, how many result in a TLB miss? Clearly indicate the accesses that result in a miss. Assume that the TLB cache is empty before the accesses begin.

Ans: 0,1,2, (miss) 1,2 (hit), 3 (miss)

(b) Out of the 6 memory accesses, how many result in a page fault? Clearly indicate the accesses that result in a page fault.

Ans: last access 3 result in a page fault

(c) Upon accessing the logical address 60, which physical address is eventually accessed by the system (after servicing any page faults that may arise)? Show suitable calculations.

Ans: 60 = 0011 1100 = page 3. 3 causes page fault, replaces LRU page 0, and mapped to frame 2. So physical address = 0010 1100 = 44

33. Consider a 64-bit system running an OS that uses hierarchical page tables to manage virtual memory. Assume that logical and physical pages are of size 4KB and each page table entry is 4 bytes in size.

(a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?

(b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.

(c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

Ans

(a) ceil (64 - 12)/(12 - 2) = 6

(b) 2, 10, 10, 10, 10, 10 (starting from most significant to least)

(c) Innermost level has 2^{52} PTEs, which fit in 2^{42} pages. The next level has 2^{42} PTEs which require 2^{32} pages, and so on. Total pages = $2^{42} + 2^{32} + 2^{22} + 2^{12} + 2^2 + 1$

34. The page size in a system (running a Linux-like operating sytem on x86 hardware) is increased while keeping everything else (including the total size of main memory) the same. For each of the following metrics below, indicate whether the metric is *generally* expected to increase, decrease, or not change as a result of this increase in page size.

    (a) Size of the page table of a process

    (b) TLB hit rate

    (c) Internal fragmentation of main memory

    Ans: (a) PT size decreases (fewer entries) (b) TLB hit rate increases (more coverage) (c) Internal fragmentation increases (more space wasted in a page)

35. Consider a process with 4 logical pages, numbered 0–3. The page table of the process consists of the following logical page number to physical frame number mappings: (0, 11), (1, 35), (2, 3), (3, 1). The process runs on a system with 16 bit virtual addresses and a page size of 256 bytes. You are given that this process accesses virtual address 770. Answer the following questions, showing suitable calculations.

    (a) Which logical page number does this virtual address correspond to?

    (b) Which physical address does this virtual address translate to?

    Ans: (a) 770 = 512 + 256 + 2 = 00000011 00000010 = page 3, offset 2

    (b) page 3 maps to frame 1. physical address = 0000001 00000010 = 256 + 2 = 258

36. Consider a system with 16 bit virtual addresses, 256 byte pages, and 4 byte page table entries. The OS builds a multi-level page table for each process. Calculate the maximum number of pages required to store all levels of the page table of a process in this system.

    Ans: Number of PTE per process = $2^{16}/2^8 = 2^8$. Number of PTE per page = $2^8/2^2 = 2^6$. Number of inner page table pages = $2^8/2^6$ = 4, which requires one outer page directory. So total pages = 4+1 = 5.

37. Consider a process with 4 physical pages numbered 0–3. The process accesses pages in the following sequence: 0, 1, 0, 2, 3, 3, 0, 2. Assume that the RAM can hold only 3 out of these 4 pages, is initially empty, and there is no other process executing on the system.

    (a) Assuming the demand paging system is using an LRU replacement policy, how many page faults do the 8 page accesses above generate? Indicate the accesses which cause the faults.

    (b) What is the minimum number of page faults that would be generated by an optimal page replacement policy? Indicate the accesses which cause the faults.

Ans:

(a) 0 (M), 1 (M), 0(H), 2 (M), 3 (M), 3(H), 0(H), 2(H) = 4 misses

(b) Same as above

38. Consider a Linux-like operating system running on a 48-bit CPU hardware. The OS uses hierarchical paging, with 8 KB pages and 4 byte page table entries.

    (a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?

    Ans: ceil (48 - 13)/(13 - 2) = 4

    (b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.

    Ans: 2, 11, 11, 11

    (c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

    Ans: Innermost level has 2^{35} PTEs. Each page can accommodate 2^{11} PTEs. Total pages = $2^{24} + 2^{13} + 2^2 + 1$

39. Consider the scenario described in the previous question. You are told that the OS uses demand paging. That is, the OS allocates a physical frame and a corresponding PTE in the page table only when the memory location is accessed for the first time by a process. Further, the pages at all levels of the hierarchical page table are also allocated on demand, i.e., when there is at least one valid PTE within that page. A process in this system has accessed memory locations in 4K unique pages so far. You may assume that none of these 4K pages has been swapped out yet. You are required to compute the minimum and maximum possible sizes of the page table of this process after all accesses have completed.

    (a) What is the minimum possible size (in pages) of the page table of this process?

    Ans: Each page holds 2^{11} PTEs. So 2^{12} pages can be accommodated in 2 pages at the inner most level. Minimum pages in each of the outer levels is 1. So minimum size = 2 + 1 + 1 + 1 = 5.

    (b) What is the maximum possible size (in pages) of the page table of this process?

    Ans: The 2^{12} pages/PTEs could have been widely spread apart and in distinct pages at all levels of the page table. So maximum size = $2^{12} + 2^{12} + 2^2 + 1$.

40. In a demand paging system, it is intuitively expected that increasing the number of physical frames will naturally lead to a reduction in the rate of page faults. However, this intuition does not hold for some page replacement policies. A replacement policy is said to suffer from *Belady's anomaly* if increasing the number of physical frames in the system can sometimes lead to an increase in the number of page faults. Consider two page replacement policies studied in class: FIFO and LRU. For each of these two policies, you must state if the policy can suffer from Belady's anomaly (yes/no). Further, if you answer yes, you must provide an example of the occurrence of the anomaly, where increasing the number of physical frames actually leads to an increase in the number of page faults. If you answer no, you must provide an explanation of why you think the anomaly can never occur with this policy.

Hint: you may consider the following example. A process has 5 logical pages, and accesses them in this order: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. You may find this scenario useful in finding an example of Belady's anomaly. Of course, you may use any other example as well.

(a) FIFO

Ans: Yes. For string above, 9 faults with 3 frames and 10 faults with 4 frames.

(b) LRU

Ans: No. The N most recently used frames are always a subset of N+1 most recently used frames. So if a page fault occurs with N+1 frames, it must have occurred with N frames also. So page faults with N+1 frames can never be higher.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Concurrency

1. Answer yes/no, and provide a brief explanation.

   (a) Is it necessary for threads in a process to have separate stacks?

   (b) Is it necessary for threads in a process to have separate copies of the program executable?

   Ans:

   (a) Yes, so that they can have separate execution state, and run independently.

   (b) No, threads share the program executable and data.

2. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

   Ans:

   Yes, by time-sharing the CPU between threads on a single core.

3. Consider a multithreaded webserver running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

   Ans: When $M < N$ and the workload to the server is CPU-bound.

4. Consider a process that uses a user level threading library to spawn 10 user level threads. The library maps these 10 threads on to 2 kernel threads. The process is executing on a 8-core system. What is the maximum number of threads of a process that can be executing in parallel?

   Ans: 2

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

   (a) Only if $M > 1$.

   (b) Only if $N \geq M$.

   (c) Only if the M kernel threads can run in parallel on a multi-core machine.

(d) User level threads should always use mutexes to protect shared data.

Ans: (d) (because user level threads can execute concurrently even on a single core)

6. Which of the following statements is/are true regarding user-level threads and kernel threads?

   (a) Every user level thread always maps to a separate schedulable entity at the kernel.

   (b) Multiple user level threads can be multiplexed on the same kernel thread

   (c) Pthreads library is used to create kernel threads that are scheduled independently.

   (d) Pthreads library only creates user threads that cannot be scheduled independently at the kernel scheduler.

   Ans: (b), (c)

7. Consider a Linux application with two threads T1 and T2 that both share and access a common variable x. Thread T1 uses a `pthread` mutex lock to protect its access to x. Now, if thread T2 tries to write to x without locking, then the Linux kernel generates a trap. [T/F]

   Ans: F

8. In a single processor system, the kernel can simply disable interrupts to safely access kernel data structures, and does not need to use any spin locks. [T/F]

   Ans: T

9. In the `pthread` condition variable API, a process calling wait on the condition variable must do so with a mutex held. State one problem that would occur if the API were to allow calls to wait without requiring a mutex to be held.

   Ans: Wakeup happening between checking for condition and sleeping causing missed wakeup.

10. Consider N threads in a process that share a global variable in the program. If one thread makes a change to the variable, is this change visible to other threads? (Yes/No)

    Ans: Yes

11. Consider N threads in a process. If one thread passes certain arguments to a function in the program, are these arguments visible to the other threads? (Yes/No)

    Ans: No

12. Consider a user program thread that has locked a pthread mutex lock (that blocks when waiting for lock to be released) in user space. In modern operating systems, can this thread be context switched out or interrupted while holding the lock? (Yes/No)

    Ans: Yes

13. Repeat the previous question when the thread holds a pthread spinlock in user space.

    Ans: Yes

14. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process be interrupted by external hardware before it releases the spinlock? (Yes/No)

    Ans: No

15. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process initiate a disk read before it releases the spinlock? (Yes/No)

    Ans: No

16. When a user space process executes the wakeup/signal system call on a pthread condition variable, does it always lead to an immediate context switch of the process that calls signal (immediately after the signal instruction)? (Yes/No)

    Ans: No

17. Consider a process in kernel mode that acquires a spinlock. For correct operation, it must disable interrupts on its CPU core for the duration that the spinlock is held, in both single core and multicore systems. [T/F]

    Ans. T

18. Consider a process in kernel mode that acquires a spinlock in a multicore system. For correct operation, we must ensure that no other kernel-mode process running in parallel on another core will request the same spinlock. [T/F]

    Ans. F

19. Multiple threads of a program must use locks when accessing shared variables even when executing on a single core system. [T/F]

    Ans: T

20. Recall that the atomic instruction compare-and-swap (CAS) works as follows:
    `CAS(&var, oldval, newval)` writes `newval` into `var` and returns true if the old value of `var` is `oldval`. If the old value of `var` is not `oldval`, `CAS` returns false and does not change the value of the variable. Write code for the function to acquire a simple spinlock using the `CAS` instruction.

    Ans: while(!CAS(&lock, 0, 1));

21. The simple spinlock implementation studied in class does not guarantee any kind of fairness or FIFO order amongst the threads contending for the spin lock. A ticket lock is a spinlock implementation that guarantees a FIFO order of lock acquisition amongst the threads contending for the lock. Shown below is the code for the function to acquire a ticket lock. In this function, the variables `next_ticket` and `now_serving` are both global variables, shared across all threads, and initialized to 0. The variable `my_ticket` is a variable that is local to a particular thread, and is not shared across threads. The atomic instruction `fetch_and_increment(&var)` atomically adds 1 to the value of the variable and returns the old value of the variable.

    ```
    acquire():
      my_ticket = fetch_and_increment(&next_ticket)
      while(now_serving != my_ticket); //busy wait
    ```

    You are now required to write the code to release the spinlock, to be executed by the thread holding the lock. Your implementation of the release function must guarantee that the next contending

thread (in FIFO order) will be able to acquire the lock correctly. You must not declare or use any other variables.

```
release(): //your code here
```

Ans:

```
release(): //your code here
now_serving++;
```

22. Consider a multithreaded program, where threads need to aquire and hold multiple locks at a time. To avoid deadlocks, all threads are mandated to use the function `acquire_locks`, instead of acquiring locks independently. This function takes as arguments a variable sized array of pointers to locks (i.e., addresses of the lock structure), and the number of lock pointers in the array, as shown in the function prototype below. The function returns once all locks have been successfully acquired.

```
void acquire_locks(struct lock *la[], int n);
//i-th lock in array can be locked by calling lock(la[i])
```

Describe (in English, or in pseudocode) one way in which you would implement this function, while ensuring that no deadlocks happen during lock acquisition. Your solution must not use any other locks beyond those provided as input. Note that multiple threads can invoke this function concurrently, possibly with an overlapping set of locks, and the lock pointers can be stored in the array in any arbitrary order. You may assume that the locks in the array are unique, and there are no duplicates within the input array of locks.

Ans. Sort locks by address struct lock *, and acquire in sorted order.

23. Consider a process where multiple threads share a common Last-In-First-Out data structure. The data structure is a linked list of "struct node" elements, and a pointer "top" to the top element of the list is shared among all threads. To push an element onto the list, a thread dynamically allocates memory for the struct node on the heap, and pushes a pointer to this struct node in to the data structure as follows.

```
void push(struct node *n) {
n->next = top;
top = n;
}
```

A thread that wishes to pop an element from the data structure runs the following code.

```
struct node *pop(void) {
struct node *result = top;
if(result != NULL) top = result->next;
return result;
}
```

A programmer who wrote this code did not add any kind of locking when multiple threads concurrently access this data structure. As a result, when multiple threads try to push elements onto this structure concurrently, race conditions can occur and the results are not always what one would expect. Suppose two threads T1 and T2 try to push two nodes n1 and n2 respectively onto the data structure at the same time. If all went well, we would expect the top two elements of the data structure would be n1 and n2 in some order. However, this correct result is not guaranteed when a race condition occurs.

Describe how a race condition can occur when two threads simultaneously push two elements onto this data structure. Describe the exact interleaving of executions of T1 and T2 that causes the race condition, and illustrate with figures how the data structure would look like at various phases during the interleaved execution.

Ans: One possible race condition is as follows. n1's next is set to top, then n2's next is set to top. So both n1 and n2 are pointing to the old top. Then top is set to n1 by T1, and then top is set to n2 by T2. So, finally, top points to n2, and n2's next points to old top. But now, n1 is not accessible by traversing the list from top, and n1 remains on a side branch of the list.

24. Consider the following scenario. A town has a very popular restaurant. The restaurant can hold N diners. The number of people in the town who wish to eat at the restaurant, and are waiting outside its doors, is much larger than N. The restaurant runs its service in the following manner. Whenever it is ready for service, it opens its front door and waits for diners to come in. Once N diners enter, it closes its front door and proceeds to serve these diners. Once service finishes, the backdoor is opened and the diners are let out through the backdoor. Once all diners have exited, another batch of N diners is admitted again through the front door. This process continues indefinitey. The restaurant does not mind if the same diner is part of multiple batches.

    We model the diners and the restaurant as threads in a multithreaded program. The threads must be synchronized as follows. A diner cannot enter until the restaurant has opened its front door to let people in. The restaurant cannot start service until N diners have come in. The diners cannot exit until the back door is open. The restaurant cannot close the backdoor and prepare for the next batch until all the diners of the previous batch have left.

    Below is given unsynchronized pseudocode for the diner and restaurant threads. Your task is to complete the code such that the threads work as desired. Please write down the complete synchronized code of each thread in your solution.

    You are given the following variables (semaphores and initial values, integers) to use in your solution. The names of the variables must give you a clue about their possible usage. You must not use any other variable in your solution.

    ```
    sem (init to 0): entering_diners, exiting_diners, enter_done, exit_done
    sem (init to 1): mutex_enter, mutex_exit
    Integer counters (init to 0): count_enter, count_exit
    ```

    All changes to the counters and other variables must be done by you in your solution. None of the actions performed by the unsynchronized code below will modify any of the variables above.

    (a) Unsynchronized code for the restaurant thread is given below. Add suitable synchronization in your solution in between these actions of the restaurant.

    ```
    openFrontDoor()
    closeFrontDoor()
    serveFood()
    openBackDoor()
    closeBackDoor()
    ```

    (b) Unsynchronized code for the diner thread is given below. Add suitable synchronization in your solution around these actions of the diner.

    ```
    enterRestaurant()
    eat()
    exitRestaurant()
    ```

Ans: Correct code for restautant thread:

```
openFrontDoor()
do N times: up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
do N times: up(exiting_diners)
down(exit_done)
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
  count_enter++
  if(count_enter == N) {
    up(enter_done)
    count_enter = 0
    }
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
  count_exit++
  if(count_exit == N) {
    up(exit_done)
    count_exit = 0
    }
up(mutex_exit)
```

An alternate to doing up N times in restaurant thread is: restaurant does up once, and every woken up diner does up once until N diners are done. This alternate solution is shown below. Correct code for restautant thread:

```
openFrontDoor()
up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
up(exiting_diners)
down(exit_done)
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
  count_enter++

  if(count_enter < N)
      up(entering_diners)
  else if(count_enter == N) {
    up(enter_done)
    count_enter = 0
    }
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
  count_exit++
  if(count_exit < N)
     up(exiting_diners)
  else if(count_exit == N) {
    up(exit_done)
    count_exit = 0
    }
up(mutex_exit)
```

25. Consider a scenario where a bus picks up waiting passengers from a bus stop periodically. The bus has a capacity of K. The bus arrives at the bus stop, allows up to K waiting passengers (fewer if less than K are waiting) to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board, and should wait for the next time the bus arrives. The bus and passengers are represented by threads in a program. The passenger thread should call the function board() after the passenger has boarded and the bus should invoke depart() when it has boarded the desired number of passengers and is ready to depart.

    The threads share the following variables, none of which are implicitly updated by functions like board() or depart().

```
mutex = semaphore initialized to 1.
bus_arrived = semaphore initialized to 0.
passenger_boarded = semaphore initialized to 0.
waiting_count = integer initialized to 0.
```

    Below is given synchronized code for the passenger thread. You should not modify this in any way.

```
down(mutex)
waiting_count++
up(mutex)
down(bus_arrived)
board()
up(passenger_boarded)
```

    Write down the corresponding synchronized code for the bus thread that achieves the correct behavior specified above. The bus should board the correct number of passengers, based on its capacity and the number of those waiting. The bus should correctly board these passengers by calling up/down on the semaphores suitably. The bus code should also update waiting_count as required. Once boarding completes, the bus thread should call depart(). You can use any extra local variables in the code of the bus thread, like integers, loop indices and so on. However, you must not use any other extra synchronization primitives.

    Ans:

```
down(mutex)
N = min(waiting_count, K)
for i= 1 to N
  up(bus_arrived)
  down(passenger_boarded)
waiting_count = waiting_count - N
up(mutex)
depart()
```

26. Consider a roller coaster ride at an amusement park. The ride operator runs the ride only when there are exactly N riders on it. Multiple riders arrive at the ride and queue up at the entrance of the ride. The ride operator waits for N riders to accumulate, and may even take a nap as he waits. Once N riders have arrived, the riders call out to the operator indicating they are ready to go on the ride. The operator then opens the gate to the ride and signals exactly N riders to enter the ride. He then waits until these N riders enter the ride, and then proceeds to start the ride.

    We model the operator and riders as threads in a program. You must write pseudocode for the operator and rider threads to enable the behavior described above. Shown below is the skeleton code for the operator and rider threads. Complete the code to achieve the behavior described above. You can assume that the functions to open, start, and enter ride are implemented elsewhere, and these functions do what the names say they do. You must write the synchronization logic around these functions in order to invoke these functions at the appropriate times. You must use only locks and condition variables for synchronization in your solution. You may declare, initialize, and use other variables (counters etc.) as required in your solution.

```
//operator code, fill in the missing details

....
open_ride()
....
start_ride()
....

//rider thread, fill in the missing details
....
enter_ride()
....
```

Ans:

```
//variables: int rider_count (initialized to 0)
//variables: int enter_count (initialized to 0)
//condvar cv_rider, cv_operator1, cv_operator2
//mutex

//operator
lock(mutex)
while(rider_count < N) wait(cv_operator1, mutex)

open_ride()
do N times: signal(cv_rider)
while(enter_count < N) wait(cv_operator2, mutex)

start_ride()
unlock(mutex)

//rider
lock(mutex)
rider_count++
if(rider_count == N) signal(cv_operator1)
wait(cv_rider, mutex) // all wait, even N-th guy

enter_ride()

enter_count++
if(enter_count == N) signal(cv_operator2)
unlock(mutex)
```

27. A host of a party has invited $N > 2$ guests to his house. Due to fear of Covid-19 exposure, the host does not wish to open the door of his house multiple times to let guests in. Instead, he wishes that all N guests, even though they may arrive at different times to his door, wait for each other and enter the house all at once. The host and guests are represented by threads in a multi-threaded program. Given below is the pseudocode for the host thread, where the host waits for all guests to arrive, then calls openDoor(), and signals a condition variable once. You must write the corresponding code for the guest threads. The guests must wait for all N of them to arrive and for the host to open the door, and must call enterHouse() only after that. You must ensure that all N waiting guests enter the house after the door is opened. You must use only locks and condition variables for synchronization.

    The following variables are used in this solution: lock m, condition variables cv_host and cv_guest, and integer guest_count (initialized to 0). You must not use any other variables in the guest for synchronization.

```
//host
lock(m)
while(guest_count < N)
  wait(cv_host, m)
openDoor()
signal(cv_guest)
unlock(m)
```

    Ans:

```
//guest
lock(m)
guest_count++
if(guest_count == N)
  signal(cv_host)
wait(cv_guest, m)
signal(cv_guest)
unlock(m)
enterHouse()
```

28. Consider the classic readers-writers synchronization problem described below. Several processes/threads wish to read and write data shared between them. Some processes only want to read the shared data ("readers"), while others want to update the shared data as well ("writers"). Multiple readers may concurrently access the data safely, without any correctness issues. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. While it is possible for each reader and writer to acquire a regular mutex and operate in perfect mutual exclusion, such a solution will be missing out on the benefits of allowing multiple readers to read at the same time without waiting for other readers to finish. Therefore, we wish to have special kind of locks called reader-writer locks that can be acquired by processes/threads in such situations. These locks have separate lock/unlock functions, depending on whether the thread asking for a lock is a reader or writer. If one reader asks for a lock while another reader already has it, the second reader will also be granted a read lock (unlike in the case of a regular mutex), thus encouraging more concurrency in the application.

    Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and write-Unlock that are invoked by the readers and writers to realize reader-writer locks. You must use condition variables and mutexes only in your solution.

    Ans: A boolean variable `writer_present`, and two condition variables, `reader_can_enter` and `writer_can_enter`, are used.

```
readLock:
lock(mutex)
while(writer_present)
  wait(reader_can_enter)
read_count++
unlock(mutex)

readUnlock:
lock(mutex)
read_count--
if(read_count==0)
  signal(writer_can_enter)
unlock(mutex)

writeLock:
lock(mutex)
while(read_count > 0 || writer_present)
  wait(writer_can_enter)
writer_present = true
unlock(mutex)

writeUnlock:
lock(mutex)
writer_present = false
signal(writer_can_enter)
signal_broadcast(reader_can_enter)
unlock(mutex)
```

29. Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use **only** semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation `down(x)` and `up(x)` to invoke atomic down and up operations on a semaphore `x` that are available via the OS API. Use sensible names for your variables.

Ans:

```
sem lock = 1; sem writer_can_enter = 1; int readCount = 0;

readLock:
down(lock)
readCount++
if(readCount ==1)
  down(writer_can_enter) //don't coexist with a writer
up(lock)

readUnlock:
down(lock)
readCount--
if(readCount == 0)
  up(writer_can_enter)
up(lock)

writeLock:
down(writer_can_enter)

writeUnlock:
up(writer_can_enter)
```

30. Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving **preference to writers**, where no waiting writer should be kept waiting for longer than necessary. For example, suppose reader process R1 is actively reading. And a writer process W1 and reader process R2 arrive while R1 is reading. While it might be fine to allow R2 in, this could prolong the waiting time of W1 beyond the absolute minimum of waiting until R1 finishes. Therefore, if we want writer preference, R2 should not be allowed before W1. Your goal is to write down pseudocode for read lock, read unlock, write lock, and write unlock functions that the processes should call, in order to realize read/write locks with writer preference. You must use only simple locks/mutexes and conditional variables in your solution. Please pick sensible names for your variables so that your solution is readable.

    Ans:

```
readLock:
lock(mutex)
while(writer_present || writers_waiting > 0)
    wait(reader_can_enter,mutex)
readcount++
unlock(mutex)

readUnlock:
lock(mutex)
readcount--
if(readcount==0)
    signal(writer_can_enter)
unlock(mutex)

writeLock:
lock(mutex)
writer_waiting++
while(readcount > 0 || writer_present)
    wait(writer_can_enter, mutex)
writer_waiting--
writer_present = true
unlock(mutex)

writeUnlock:
lock(mutex)
writer_present = false
if(writer_waiting==0)
    signal_broadcast(reader_can_enter)
else
    signal(writer_can_enter)
unlock(mutex)
```

31. Write a solution to the readers-writers problem with preference to writers discussed above, but using only semaphores.

Ans:

```
sem rlock = 1; sem wlock = 1;
sem reader_can_try = 1; sem writer_can_enter = 1;
int readCount = 0; int writeCount = 0;

readLock:
down(reader_can_try) //new sem blocks reader if writer waiting
down(rlock)
readCount++
if(readCount ==1)
  down(writer_can_enter) //don't coexist with a writer
up(rlock)
up(reader_can_try)

readUnlock:
down(rlock)
readCount--
if(readCount == 0)
  up(writer_can_enter)
up(rlock)

writeLock:
down(wlock)
writerCount++
if(writerCount==1)
  down(reader_can_try)
up(wlock)
down(writer_can_enter)  //release wlock and then block

writeUnlock:
down(wlock)
writerCount--
if(writerCount == 0)
  up(reader_can_try)
up(wlock)

up(writer_can_enter)
```

32. Consider the famous dining philosophers' problem. N philosophers are sitting around a table with N forks between them. Each philosopher must pick up both forks on her left and right before she can start eating. If each philosopher first picks the fork on her left (or right), then all will deadlock while waiting for the other fork. The goal is to come up with an algorithm that lets all philosophers eat, without deadlock or starvation. Write a solution to this problem using condition variables.

    Ans: A variable `state` is associated with each philosopher, and can be one of EATING (holding both forks) or THINKING (when not eating). Further, a condition variable is associated with each philosopher to make them sleep and wake them up when needed. Each philosopher must call the `pickup` function before eating, and `putdown` function when done. Both these functions use a mutex to change states only when both forks are available.

```
bothForksFree(i):
return (state[leftNbr(i)] != EATING &&
        state[rightNbr(i)] != EATING)

pickup(i):
  lock(mutex)
  while(!bothForksFree(i))
      wait(condvar[i])
  state[i] = EATING
  unlock(mutex)

putdown(i):
  lock(mutex)
  state[i] = THINKING
  if(bothForksFree(leftNbr(i)))
      signal(leftNbr(i))
  if(bothForksFree(rightNbr(i)))
      signal(rightNbr(i))
  unlock(mutex)
```

33. Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly, the the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient "enters the doctors office", she conveys her symptoms to the doctor using a call to `consultDoctor()`, which updates the shared memory with the patient's symptoms. The doctor then calls `treatPatient()` to access the buffer and update it with details of the treatment. Finally, the patient process must call `noteTreatment()` to see the updated treatment details in the shared buffer, before leaving the doctor's office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use **only semaphores** to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

    Ans:

    (a) Semaphores variables:

    ```
    pt_waiting = 0
    treatment_done = 0
    doc_avlbl = 1
    ```

    (b) Patient process:

    ```
    down(doc_avlbl)
    consultDoctor()
    up(pt_waiting)
    down(treatment_done)
    noteTreatment()
    up(doc_avlbl)
    ```

    (c) Doctor:

    ```
    while(1) {
    down(pt_waiting)
    treatPatient()
    up(treatment_done)
    }
    ```

34. Consider a multithreaded banking application. The main process receives requests to tranfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type `mylock` (much like a `pthreads` mutex) as shown below.

```
struct account {
int accountnum;
int balance;
mylock lock;
};
```

Each thread that receives a transfer request must implement the transfer function shown below, which transfers money from one account to the other. Add correct locking (by calling the `dolock(&lock)` and `unlock(&lock)` functions on a `mylock` variable) to the tranfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

```
void transfer(struct account *from, struct account *to, int amount) {

from->balance -= amount; // dont write anything...
to->balance += amount; // ...between these two lines

}
```

Ans: The accounts must be locked in order of their account numbers. Otherwise, a transfer from account X to Y and a parallel transfer from Y to X may acquire locks on X and Y in different orders and end up in a deadlock.

```
struct account *lower = (from->accountnum < to->accountnum)?from:to;
struct account *higher = (from->accountnum < to->accountnum)?to:from;
dolock(&(lower->lock));
dolock(&(higher->lock));

from->balance -= amount;
to->balance += amount;

unlock(&(lower->lock));
unlock(&(higher->lock));
```

35. Consider a process with three threads A, B, and C. The default thread of the process receives multiple requests, and places them in a request queue that is accessible by all the three threads A, B, and C. For each request, we require that the request must first be processed by thread A, then B, then C, then B again, and finally by A before it can be removed and discarded from the queue. Thread A must read the next request from the queue only after it is finished with all the above steps of the previous one. Write down code for the functions run by the threads A, B, and C, to enable this synchronization. You can only worry about the synchronization logic and ignore the application specific processing done by the threads. You may use any synchronization primitive of your choice to solve this question.

    Ans: Solution using semaphores shown below. The order of processing is A1–B1–C–B2–A2. All threads run in a forever loop, and wait as dictated by the semaphores.

```
sem a1done = 0; b1done = 0; cdone = 0; b2done = 0;

ThreadA:
  get request from queue and process
  up(a1done)
  down(b2 done)
  finish with request

ThreadB:
  down(a1done)
  //do work
  up(b1done)
  down(cdone)
  //do work
  up(b2done)

ThreadC:
  down(b1done)
  //do work
  up(cdone)
```

36. Consider two threads A and B that perform two operations each. Let the operations of thread A be A1 and A2; let the operations of thread B be B1 and B2. We require that threads A and B each perform their first operation before either can proceed to the second operation. That is, we require that A1 be run before B2 and B1 before A2. Consider the following solutions based on semaphores for this problem (the code run by threads A and B is shown in two columns next to each other). For each solution, explain whether the solution is correct or not. If it is incorrect, you must also point out why the solution is incorrect.

(a)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A              //Thread B
A1                      B1
down(B1Done)            down(A1Done)
up(A1Done)              up(B1Done)
A2                      B2
```

(b)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A              //Thread B
A1                      B1
down(B1Done)            up(B1Done)
up(A1Done)              down(A1Done)
A2                      B2
```

(c)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A              //Thread B
A1                      B1
up(A1Done)              up(B1Done)
down(B1Done)            down(A1Done)
A2                      B2
```

Ans:

(a) Deadlocks, so incorrect.

(b) Correct

(c) Correct

37. Now consider a generalization of the above problem for the case of N threads that want to each execute their first operation before any thread proceeds to the second operation. Below is the code that each thread runs in order to achieve this synchronization. `count` is an integer shared variable, and `mutex` is a mutex binary semaphore that protects this shared variable. `step1Done` is a semaphore initialized to zero. You are told that this code is wrong and does not work correctly. Further, you can fix it by changing it slightly (e.g., adding one statement, or rearranging the code in some way). Suggest the change to be made to the code in the snippet below to fix it. You must use only semaphores and no other synchronization mechanism.

```
//run first step

down(mutex);
count++;
up(mutex);
if(count == N)
       up(step1Done);
down(step1Done);

//run second step
```

Ans: The problem is that the semaphore is decremented N times, but is only incremented once. To fix it, we must do up N times when count is N. Or, add up after the last down, so that it is performed N times by the N threads.

38. The cigarette smokers problem is a classical synchronization problem that involves 4 threads: one agent and three smokers. The smokers require three ingredients to smoke a cigarette: tobacco, paper, and matches. Each smoker has one of the three ingredients and waits for the other two, smokes the cigar once he obtains all ingredients, and repeats this forever. The agent repeatedly puts out two ingredients at a time and makes them available. In the correct solution of this problem, the smoker with the complementary ingredient should finish smoking his cigar. Consider the following solution to the problem. The shared variables are three semaphores `tobacco`, `paper` and `matches` initialized to 0, and semaphore `doneSmoking` initialized to 1. The agent code performs `down(doneSmoking)`, then picks two of the three ingredients at random and performs `up` on the corresponding two semaphores, and repeats. The smoker with tobacco runs the following code in a loop.

```
down(paper)
down(matches)
//make and smoke cigar
up(doneSmoking)
```

Similarly, the smoker with matches waits for tobacco and paper, and the smoker with paper waits for tobacco and matches, before signaling the agent that they are done smoking. Does the code above solve the synchronization problem correctly? If you answer yes, provide a justification for why the code is correct. If you answer no, describe what the error is and also provide a correct solution to the problem. (If you think the code is incorrect and are providing another solution, you may change the code of both the agent and the smokers. You can also introduce new variables as necessary. You must use only semaphores to solve the problem.)

Ans: The code is incorrect and deadlocks. One fix is to add semaphores for two ingredients at a time (e.g., tobaccoAndPaper). The smokers wait on these and the agent signals these. So there is no possibility of deadlock.

39. Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients. For every buy or sell request received by the server, the main process spawns a new buy or sell thread. We require that every buy thread waits until a sell thread arrives, and vice versa. A matched pair of buy and sell threads will both return a response to the clients and exit. You may assume that all buy/sell requests are identical to each other, so that any buy thread can be matched with any sell thread. The code executed by the buy thread is shown below (the code of the sell thread would be symmetric). You have to write the synchronization logic that must be run at the start of the execution of the thread to enable it to wait for a matching sell thread to arrive (if none exists already). Once the threads are matched, you may assume that the function `completeBuy()` takes care of the application logic for exchanging information with the matching thread, communicating with the client, and finishing the transaction. You may use any synchronization technique of your choice.

```
//declare any variables here

buy_thread_function:
  //start of sync logic


  //end of sync logic
  completeBuy();
```

Ans:

```
sem buyer = 0; sem seller = 0;

Buyer thread:

up(buyer)
down(seller)
completeBuy()
```

40. Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs and awaits his turn. The barber moves onto the next waiting seated customer after he finishes one hair cut. If there are no customers to be served, the barber goes to sleep. If the barber is asleep when a customer arrives, the customer wakes up the barber to give him a hair cut. A waiting customer vacates his chair after his hair cut completes. Your goal is to write the pseudocode for the customer and barber threads below with suitable synchronization. You must use only semaphores to solve this problem. Use the standard notation of invoking up/down functions on a semaphore variable.

    The following variables (3 semaphores and a count) are provided to you for your solution. You must use these variables and declare any additional variables if required.

    ```
    semaphore mutex = 1, customers = 0, barber = 0;
    int waiting_count = 0;
    ```

    Some functions to invoke in your customer and barber threads are:

    - A customer who finds the waiting room full should call the function `leave()` to exit the shop permanently. This function does not return.
    - A customer should invoke the function `getHairCut()` in order to get his hair cut. This function returns when the hair cut completes.
    - The barber thread should call `cutHair()` to give a hair cut. When the barber invokes this function, there should be exactly one customer invoking `getHairCut()` concurrently.

Ans:

```
Customer:

down(mutex)
if(waiting_count == N)
  up(mutex)
  leave()
waiting_count++
up(mutex)

up(customers)
down(barber)

getHairCut()

down(mutex)
waiting_count--
up(mutex)

Barber:

up(barber)
down(customers)
cutHair()
```

41. Consider a multithreaded application server handling requests from clients. Every new request that arrives at the server causes a new thread to be spawned to handle that request. The server can provide service to only one request/thread at a time, and other threads that arrive when the server is busy must wait for service using a synchronization primitive (semaphore or condition variable). In order to avoid excessive waiting times, the server does not wish to have more than N requests/threads in the system (including the waiting requests and any request it is currently serving). You may assume that $N > 2$. Given this constraint, a newly arriving thread must first check if N other requests are already in the system: if yes, it must exit without waiting and return an error value to the client, by calling the function `thr_exit_failure()`. This function terminates the thread and does not return.

    When a thread is ready for service, it must call the function `get_service()`. Your code should ensure that no more than one thread calls this function at any point of time. This function blocks the thread for the duration of the service. Note that, while the thread receiving service is blocked, other arriving threads must be free to join the queue, or exit if the system is overloaded. After a thread returns from `get_service()`, it must enable one of the waiting threads to seek service (if any are waiting), and then terminate itself succesfully by calling the function `thr_exit_success()`. This function terminates the thread and does not return.

    You are required to write pseudocode of the function to be run by the request threads in this system, as per the specification above. Your solution must use only locks and condition variables for synchronization. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
int num_requests=0;
bool server_busy = false
cv, mutex

lock(mutex)

if(num_requests == N)
  unlock(mutex)
  the_exit_failure()

num_requests++

if(server_busy)
  wait(cv, mutex)

server_busy = true
unlock(mutex)

get_service()

lock(mutex)
num_requests--
server_busy = false

if(num_requests > 0)
  signal(cv)

unlock(mutex)
thr_exit_success()
```

42. Consider the previous problem, but now assume that N is infinity. That is, all arriving threads will wait (if needed) for their turn in the queue of a synchronization primitive, get served when their turn comes, and exit successfully. Write the pseudocode of the function to be run by the threads with this modified specification. Your solution must only use semaphores for synchronization, and only the correct solution that uses the least number of semaphores will get full credit. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
sem waiting = 1

down(waiting)
get_service()
up(waiting)
thr_exit_success()
```

43. Consider the following synchronization problem. A group of children are picking chocolates from a box that can hold up to N chocolates. A child that wants to eat a chocolate picks one from the box to eat, unless the box is empty. If a child finds the box to be empty, she wakes up the mother, and waits until the mother refills the box with N chocolates. Unsynchronized code snippets for the child and mother threads are as shown below:

```
//Child
while True:
  getChocolateFromBox()
  eat()

//Mother
while True:
  refillChocolateBox(N)
```

You must now modify the code of the mother and child threads by adding suitable synchronization such that a child invokes `getChocolateFromBox()` only if the box is non-empty, and the mother invokes `refillChocolateBox(N)` only if the box is fully empty. Solve this question using only locks and condition variables, and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;
mutex m; // you may invoke lock and unlock
condvar fullBox, emptyBox; //you may perform wait and signal
                           //or signal_broadcast
```

(a) Code for child thread

(b) Code for mother thread

Ans:

```
//Child
while True:
  lock(m)
  while(count == 0)
    signal(emptyBox)
    wait(fullBox, m)
  getChocolateFromBox()
  eat()
  count--
  signal(fullBox) //optional
  unlock(m)

//Mother
while True:
  lock(m)
  if(count > 0)
    wait(emptyBox, m)
  refillChocolateBox(N)
  count += N
  signal(fullBox)
  unlock(m)
```

There are two ways of waking up sleeping children. Either the mother does a signal broadcast to all children. Or every child that eats a chocolate wakes up another sleeping child. You may also assume that signal by mother wakes up all children.

44. Repeat the above question, but your solution now must use only semaphores and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;
semaphore m, fullBox, emptyBox;
//initial values of semaphores are not specified
//you may invoke up and down methods on a semaphore
```

(a) Initial values of the semaphores

(b) Code for child thread

(c) Code for mother thread

Ans:

```
m = 1, fullBox = 0, emptyBox = 0

//Child
while True:
  down(m)
  if(count == 0)
    up(emptyBox)
    down(fullBox)
    count += N
  getChocolateFromBox()
  eat()
  count--
  up(m)

//Mother
while True:
  down(emptyBox)
  refillChocolateBox(N)
  up(fullBox)
```

Here the subtlety is the lock m. Mother can't get lock to update count after filling the box, as that will cause a deadlock. In general, if child sleeps with mutex m locked, then mother cannot request the same lock.

45. Consider the classic "barrier" synchronization problem, where N threads wish to synchronize with each other as follows. N threads arrive into the system at different times and in any order. The arriving threads must wait until all N threads have arrived into the system, and continue execution only after all N threads have arrived. We wish to write logic to synchronize the threads in the manner stated above using semaphores. Below are three possible solutions to the problem. You are told that one of the solutions is correct and the other two are wrong. Identify the correct solution amongst the three given options. Further, for each of the other incorrect solutions, explain clearly why the solution is wrong. The following shared variables are declared for use in each solution.

```
int count = 0;
sem mutex; //initialized to 1
sem barrier; //initialized to 0
```

(a)
```
down(mutex)
  count++
  if(count == N) up(barrier)
up(mutex)

down(barrier)

//wait done; proceed to actual task
```

(b)
```
down(mutex)
  count++
  if(count == N) up(barrier)
up(mutex)

down(barrier)
up(barrier)

//wait done; proceed to actual task
```

(c)
```
down(mutex)
  count++
  if(count == N) up(barrier)
  down(barrier)
  up(barrier)
up(mutex)

//wait done; proceed to actual task
```

Ans: In (a) up is done only once when many threads are waiting on down. In (c), down(barrier) is called when mutex held, so code deadlocks. (b) is correct answer.

46. Consider the barrier synchronization primitive discussed in class, where the N threads of an application wait until all the threads have arrived at a barrier, before they proceed to do a certain task. You are now required to write the code for a reusable barrier, where the N application threads perform a series of steps in a loop, and use the same barrier code to synchronize for each iteration of the loop. That is, your solution should ensure that all threads wait for each other before the start of each step, and proceed to the next step only after all threads have completed the previous step. Your solution must only use semaphores. The following functions can be invoked on a semaphore `s` used in this question: `down(s)`, `up(s)`, and `up(s,n)`. While the first two functions are as studied in class, the function `up(s,n)` simply invokes `up(s)` n times atomically.

    We have provided you some code to get started. Shown below is the code to be run by each application thread, including the code to wait at the barrier. However, this is not the correct solution, as this code only works as a single-use barrier, i.e., it only ensures that the threads synchronize at the barrier once, and cannot be used to synchronize multiple times (can you figure out why?). You are required to modify this code to make it reusable, such that the threads can synchronize at the barrier multiple times for the multiple steps to be performed.

    Your solution must only use the following variables: `int count = 0`; and semaphores (initial values as given): `sem mutex = 1`; `sem barrier1 = 0`; `sem barrier2 = 0`;

```
For each step to be executed by the threads, do:

//add code here if required to make barrier reusable


down(mutex)
  count++
  if(count == N) up(barrier1, N)
up(mutex)
down(barrier1)

... wait done, execute actual task of this step ...

//add code here if required to make barrier reusable for next step
```

    Ans: The extra code to be added is at the end of completing a step, where you make all threads wait once again.

```
down(mutex)
count--
if(count==0) up(barrier2, N)
up(mutex)
down(barrier2)
```

47. Consider a web server that is supposed to serve a batch of N requests. Each request that arrives at the web server spawns a new thread. The arriving threads wait until N of them accumulate, at which point all of them proceed to get service from the server. Shown below is the code executed by each arriving thread, that causes it to wait until all the other threads arrive. The variable `count` is initialized to N. The code also uses `wait` and `signal` primitives on a condition variable; and you may assume that the signal primitive wakes up all waiting threads (not just one of them).

```
lock(mutex)
  count--;
unlock(mutex)

if(count > 0) {
  lock(mutex)
  wait(cv, mutex)
  unlock(mutex)
}
else {
  lock(mutex)
  signal(cv)
  unlock(mutex)
}

... wait done, proceed to server ...
```

You are told that the code above is incorrect, and can sometimes cause a deadlock. That is, in some executions, all N threads do not go to the server for service, even though they have arrived.

(a) Using an example, explain the exact sequence of events that can cause a deadlock. You must write your answers as bullet points, with one event per bullet point, starting from threads arriving in the system until the deadlock.

(b) Explain how you will fix this deadlock and correct the code shown above. You must retain the basic structure of the code. Indicate your changes next to the code snippet above.

Ans: The given incorrect solution may cause a missed wakeup. For example, some thread decides to wait and goes inside the if-loop, but is context switched out before calling wait (and before it acquires the lock). Now, if count hits 0 and signal happens before it runs again, it will wait with no one to wake it up, leading to deadlock. The fix is simply holding the lock all through the condition checking and waiting.

48. Consider an application that has $K + 1$ threads running on a Linux-like OS ($K > 1$). The first K threads of an application execute a certain task T1, and the remaining one thread executes task T2. The application logic requires that task T1 is executed $N > 1$ times, followed by task T2 executed once, and this cycle of N executions of T1 followed by one execution of T2 continue indefinitely. All K threads should be able to participate in the N executions of task T1, even though it is not required to ensure perfect fairness amongst the threads.

    Shown below is one possible set of functions executed by the threads running tasks T1 and T2. You are told that this solution has two bugs in the code run by the thread performing task T2. Briefly describe the bugs in the space below, and suggest small changes to the corresponding code to fix these bugs (you may write your changes next to the code snippet). You must not change the code corresponding to task T1 in any way. All threads share a counter `count` (initialized to 0), a mutex variable `m`, and two condition variables `t1cv`, and `t2cv`. Here, the function `signal` on a condition variable wakes up only one of the possibly many sleeping threads.

```
//function run by K threads of task T1
while True {
    lock(m)
    if(count >= N) {
        signal(t2cv)
        wait(t1cv, m)
    }
    //.. do task T1 once ..
    count++
    unlock(m)
}
//function run by thread of task T2
while True {
    lock(m)
    wait(t2cv, m)
    // .. do task T2 once
    count = 0
    signal(t1cv)
    unlock(m)
}
```

    Ans: (a) check count<N and only then wait (b) signal broadcast instead of signal

49. You are now required to solve the previous question using semaphores for synchronization. You are given the pseudocode for the function run by the thread executing task T2 (which you must not change). You are now required to write the corresponding code executed by the K threads running task T1. You must use the following semaphores in your solution: `mutex`, `t1sem`, `t2sem`. You must initialize them suitably below. The variable `count` (initialized to 0) is also available for use in your solution.

Ans:

```
//fill in initial values of semaphores
sem_init(mutex,   );  sem_init(t1sem,   );   sem_init(t2sem,    );
//other variables
int count = 0

//function run by thread executing T2
while True {
    down(t2sem)
    //.. do task T2 ..
    up(t1sem)
}

//function run by threads executing task T1
while True {


}
```

Ans:

```
mutex=1, t1sem=0, t2sem=0

down(mutex)
if(count == N)
  up(t2sem)
  down(t1sem)
  count = 0

do task T1 once
count++
up(mutex)
```

50. Multiple people are entering and exiting a room that has a light switch. You are writing a computer program to model the people in this situation as threads in an application. You must fill in the functions `onEnter()` and `onExit()` that are invoked by a thread/person when the person enters and exits a room respectively. We require that the first person entering a room must turn on the light switch by invoking the function `turnOnSwitch()`, while the last person leaving the room must turn off the switch by invoking `turnOffSwitch()`. You must invoke these functions suitably in your code below. You may use any synchronization primitives of your choice to achieve this desired goal. You may also use any variables required in your solution, which are shared across all threads/persons.

    (a) Variables and initial values

    (b) Code `onEnter()` to be run by thread/person entering

    (c) Code `onExit()` to be run by thread/person exiting

    Ans:

```
variables: mutex, count

onEnter():
lock(mutex)
count++
if(count==1) turnOnSwitch()
unlock(mutex)

onExit():
lock(mutex)
count--
if(count==0) turnOffSwitch()
unlock(mutex)
```

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: File systems

1. Provide one reason why a DMA-enabled device driver usually gives better performance over a non-DMA interrupt-driven device driver.

   Ans: A DMA driver frees up CPU cycles that would have been spent copying data from the device to physical memory.

2. Which of the following statements is/are true regarding memory-mapped I/O?

   A. The CPU accesses the device memory much like it accesses main memory.

   B. The CPU uses separate architecture-specific instructions to access memory in the device.

   C. Memory-mapped I/O cannot be used with a polling-based device driver.

   D. Memory-mapped I/O can be used only with an interrupt-driven device driver.

   Ans: A

3. Consider a file D1/F1 that is hard linked from another parent directory D2. Then the directory entry of this file (including the filename and inode number) in directory D1 must be exactly identical to the directory entry in directory D2. [T/F]

   Ans: F (the file name can be different)

4. It is possible for a system that uses a disk buffer cache with FIFO as the buffer replacement policy to suffer from the Belady's anomaly. [T/F]

   Ans: T

5. Reading files via memory mapping them avoids an extra copy of file data from kernel space buffers to user space buffers. [T/F]

   Ans: T

6. A soft link can create a link between files across different file systems, whereas a hard link can only create links between a directory and a file within the same file system. [T/F]

   Ans: T (becasue hard link stores inode number, which is unique only within a file system)

7. Consider the process of opening a new file that does not exist (obviously, creating it during opening), via the "open" system call. Describe changes to all the in-memory and disk-based file system structures (e.g., file tables, inodes, and directories) that occur as part of this system call implementation. Write clearly, listing the structure that is changed, and the change made to it.

   Ans: (a) New inode allocated on disk (with link count=1), and inode bitmap updated in the process. (b) Directory entry added to parent directory, to add mapping from file name to inode number.

(c) In-memory inode allocated. (d) System-wide open file table points to in-memory inode. (e) Per-process file descriptor table points to open file table entry.

8. Now, suppose the process that has opened the file in the previous question proceeds to write 100 bytes into the file. Assume block size on disk is 512 bytes. Assume the OS uses a write-through disk buffer cache. List all the operations/changes to various datastructures that take place when the write operation successfully completes.

   Ans: (a) open file table offset is changed (b) in-memory and on-disk inode adds pointer to new data block, and last modified time is updated (c) a copy of the data block comes into the disk buffer cache (d) New data block is allocated from data block bitmap (e) new data block is filled with user provided data

9. Repeat the above question for the implementation of the "link" system call, when linking to an existing file (not open from any process) in a directory from another new parent directory.

   Ans: (a) The link count of the on-disk inode of the file is incremented. (b) A directory entry is added to the new directory to create a mapping from the file name to the inode number of the original file (if the new directory does not have space in its data blocks for the new file, a new data block is allocated for the new directory entry, and a pointer to this data block is added from the directory's inode).

10. Repeat the above question for the implementation of the "dup" system call on a file descriptor.

    Ans: To dup a file descriptor, another empty slot in the file descriptor table of the process is found, and this new entry is set to point to the same global open file table entry as the old file descriptor. That is, two FDs point to same system-wide file table entry.

11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

    Ans: (N+M)*512 bytes

12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

    Ans: $2^N * M$ bytes

13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

    Ans: Number of data blocks = $2^{41}/2^9 = 2^{32}$, so 32 bits or 4 bytes are required to store the number of a data block.

    Number of data block pointers in the inode = 64/4 = 16, of which 14 are direct blocks. The single indirect block stores pointers to 512/4 = 128 data blocks. The double indirect block points to 128 single indirect blocks, which in turn point to 128 data blocks each.

So, the total number of data blocks in a file can be 14 + 128 + 128*128 = 16526, and the maximum file size is 16526*512 bytes.

14. Consider a filesystem managing a disk with block size 2^b bytes, and disk block addresses of 2^a bytes. The inode of a file contains n direct blocks, one single indirect block, one double indirect block, and one triple indirect block. What is the maximum size of a file (in bytes) that can be stored in this filesystem? Assume that the indirect blocks only store a sequence of disk addresses, and no other metadata.

    Ans: Let x = number of disk addresses per block = 2^{b-a}. Then max file size is $2^b * (n + x + x^2 + x^3)$.

15. The `fork` system call creates new entries in the open file table for the newly created child process. [T/F]

    Ans: F

16. When a process opens a file that is already being read by another process, the file descriptors in both processes will point to the same open file table entry. [T/F]

    Ans: F

17. Memory mapping a file using the `mmap` system call adds one or more entries to the page table of the process. [T/F]

    Ans: T

18. The read system call to fetch data from a file always blocks the invoking process. [T/F]

    Ans: F (the data may be readily available in the disk buffer cache)

19. During filesystem operations, if the filesystem implementation ensures that changes to data blocks of a file are flushed to disk before changes to metadata blocks (like inodes and bitmaps), then the filesystem will never be in an inconsistent state after a crash, and a filesystem checker need not be run to detect and fix any inconsistencies. [T/F]

    Ans: F (If there are multiple metadata operations, some may have happened and some may have been lost, causing an inconsistency. For example, a bitmap may indicate a data block is allocated but no inode points to it.)

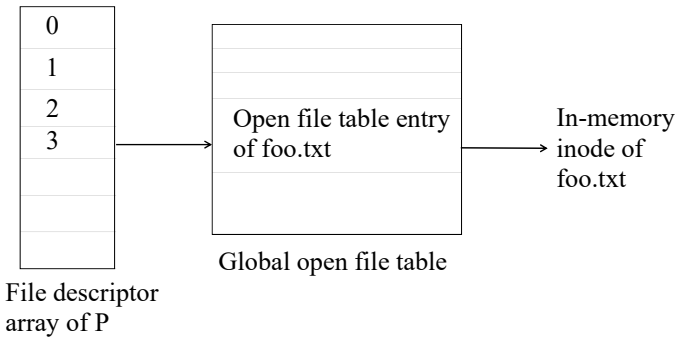
20. Interrupt-based device drivers give superior performance to polling-based drivers because they eliminate the time spent by the CPU in copying data to and from the device hardware. [T/F]

    Ans: F

21. When a process writes a block to the disk via a disk buffer cache using the write-back policy, the process invoking the write will block until the write is committed to disk. [T/F]

    Ans: F

22. Consider a process P that has opened a file `foo.txt` using the `open` system call. The figure below shows the file descriptor array of P and the global open file table, and the pointers linking these data structures.



(a) After opening the file, P forks a child C. Draw a figure showing the file descriptor arrays of P and C, and the global open file table, immediately after the fork system call successfully completes. It is enough to show the entries pertaining to the file `foo.txt`, as in the figure above.

(b) Repeat part (a) for the following scenario: after P forks a child C, another process Q also opens the same file `foo.txt`.

Ans: In (a), the file descriptor arrays of P and C are pointing to the same file table entry. In (b), the file descriptor array of Q is pointing to a new open file table entry, which points to the same inode of the file foo.txt.