**Q1. Explain the purpose of UNUSED in proc\_state. When will a process be in this state?**

**How can a parent and child process communicate?**

A parent and child can communicate through any of the normal inter-process communication schemes (pipes, sockets, message queues, shared memory), but also have some special ways to communicate that take advantage of their relationship as a parent and child. One of the most obvious is that the parent can get the exit status of the child.

**What is a zombie?**

When a program forks and the child finishes before the parent, the kernel still keeps some of its information about the child in case the parent might need it - for example, the parent may need to check the child's exit status. To be able to get this information, the parent calls `wait()'; In the interval between the child terminating and the parent calling `wait()', the child is said to be a `zombie' (If you do `ps', the child will have a `Z' in its status field to indicate this.)

**What are the process states in Unix?**

As a process executes it changes state according to its circumstances. Unix processes have the following states:  
**Running** : The process is either running or it is ready to run .  
**Waiting** : The process is waiting for an event or for a resource.  
**Stopped** : The process has been stopped, usually by receiving a signal.  
**Zombie** : The process is dead but have not been removed from the process table.

|  |
| --- |
| // Look in the process table for an UNUSED proc. |
|  | // If found, change state to EMBRYO and initialize |
|  | // state required to run in the kernel. |
|  | // Otherwise return 0. |

**Q2. What information In Fig. 4.5 [The xv6 Proc Structure] tells us that the memory allocation for a process is contiguous in physical memory, i.e., the process is not broken in pieces?**

**Contiguous storage**

A program occupies a single contiguous block of storage locations.

Faster in store

More waste in memory

**Non-contiguous storage**

A program is divided into several blocks or segments that may be placed in memory in pieces not necessary adjacent to one another.

Slower in store

Less waste in memory

**Variable Partitions** – Physical memory broken up into variable sized partitions. Physical address = virtual address + base register. Advantages – no internal fragmentation, but external fragmentation - as load and unload jobs holes left throughout physical memory.

Dealing with fragmentation stops everyone’s execution & compacts memory (adjusts base and limit registers – what causes virtual addresses to point to physical address.

**Paging** – program divided into pages of same size. Page tables translate to physical addresses. Sharing between processes by telling system want to share set of pages. Processes view memory as contiguous address space but physical mem. is non-contiguous. But virtual pages are really scattered across physical memory frames. Program cannot reference memory outside of VAS – protection Advantage: no external fragmentation. Easy to allocate physical memory (just allocated from free list of frames). Entire program need not be memory resident (in memory). User sees memory as contiguous. Disadvantages: But internal fragmentation – extra space between stack and heap (virtual AS). Extra space at end of page table (not multiple of 4K). Mem. Reference overhead, 2 references per address lookup. **Solution: use TLB – hardware cache.**

**Q3. In Fig 5.2 [Calling fork() And wait() (p2.c)], make the following changes:  
--at line 5 [the blank line after sys/wait.h>], add the statement: int counter = 0;  
--In between lines “int rc=fork();” and “if (rc < 0)…”, add the statement: counter++;  
--delete all the print statements from the program  
--Then add a statement immediately before “return 0;”  that prints the value of the counter  
What are all the possibilities regarding total output produced by this program in any run? (Keep in mind what happened in Fig 2.5. [A Multi-threaded program (threads.c)] )**

#include <stdio.h>

2 #include <stdlib.h>

3 #include <unistd.h>

4 #include <sys/wait.h>

5

6 int main(int argc, char \*argv[]) {

7 printf("hello world (pid:%d)\n", (int) getpid());

8 int rc = fork();

9 if (rc < 0) { // fork failed; exit

10 fprintf(stderr, "fork failed\n");

11 exit(1);

12 } else if (rc == 0) { // child (new process)

13 printf("hello, I am child (pid:%d)\n", (int) getpid());

14 } else { // parent goes down this path (main)

15 int rc\_wait = wait(NULL);

16 printf("hello, I am parent of %d (rc\_wait:%d) (pid:%d)\n",

17 rc, rc\_wait, (int) getpid());

18 }

19 return 0;

20 }

21

**Q4. In ch 5, fig. 5.3 [Calling fork(), wait(), And exec() (p3.c) ]:  
                In the line “ } else {  //parent goes down this path (main)”:  
delete “else {“  and   
             delete the “}” immediately before “return 0;”.  
Will the new code have the same outputs as before?**

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <unistd.h>

4 #include <string.h>

5 #include <sys/wait.h>

6

7 int main(int argc, char \*argv[]) {

8 printf("hello world (pid:%d)\n", (int) getpid());

9 int rc = fork();

10 if (rc < 0) { // fork failed; exit

11 fprintf(stderr, "fork failed\n");

12 exit(1);

13 } else if (rc == 0) { // child (new process)

14 printf("hello, I am child (pid:%d)\n", (int) getpid());

15 char \*myargs[3];

16 myargs[0] = strdup("wc"); // program: "wc" (word count)

17 myargs[1] = strdup("p3.c"); // argument: file to count

18 myargs[2] = NULL; // marks end of array

19 execvp(myargs[0], myargs); // runs word count

20 printf("this shouldn’t print out");

21 } else { // parent goes down this path (main)

22 int rc\_wait = wait(NULL);

23 printf("hello, I am parent of %d (rc\_wait:%d) (pid:%d)\n",

24 rc, rc\_wait, (int) getpid());

25 }

26 return 0;

27 }

28\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Q5. In ch 5, fig 5.4 [All of the above with redirection (p4.c)], delete the line “close…”. How will this affect the Outputs produced by the program?**

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <unistd.h>

4 #include <string.h>

5 #include <fcntl.h>

6 #include <sys/wait.h>

7

8 int main(int argc, char \*argv[]) {

9 int rc = fork();

10 if (rc < 0) { // fork failed; exit

11 fprintf(stderr, "fork failed\n");

12 exit(1);

13 } else if (rc == 0) { // child: redirect standard output to a file

14 close(STDOUT\_FILENO);

15 open("./p4.output", O\_CREAT|O\_WRONLY|O\_TRUNC, S\_IRWXU);

16

17 // now exec "wc"...

18 char \*myargs[3];

19 myargs[0] = strdup("wc"); // program: "wc" (word count)

20 myargs[1] = strdup("p4.c"); // argument: file to count

21 myargs[2] = NULL; // marks end of array

22 execvp(myargs[0], myargs); // runs word count

23 } else { // parent goes down this path (main)

24 int rc\_wait = wait(NULL);

25 }

26 return 0;

27 }

Q6. In figure 5.4 [All of the above with redirection (p4.c)], line “int rc = fork();”:   
when fork is executed, suppose the OS does not copy the image of the parent process into the child process yet, because it knows that soon this image will be replaced by the call to execvp in the line “execvp(…)”. Instead, it creates everything else about the child process (e.g., pid value, process control block, etc.). Later, when execvp is called, it makes the child’s image in memory as usual.  
This way, we are saving the time in copying the image of parent process into the child process.  
Will this work? Explain.