3.1 Using the program shown below, explain what the output will be at Line A.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
pid t pid;
pid = fork();
if (pid == 0) { /* child process */
value += 15;
return 0;
}
else if (pid > 0) { /* parent process */
wait(NULL);
printf("PARENT: value = %d",value); /* LINE A */
return 0;
}
}
```

Answer: The result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5.

3.2 Including the initial parent process, how many processes are created by the program shown below?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
/* fork a child process */
fork();
/* fork another child process */
fork();
/* and fork another */
fork();
return 0;
}
```

Answer: There are 8 processes created.

3.3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

Answer:

Concurrency in Operating Systems is when two processes are being performed at the same time. While only one process is executed at a time by the CPU these processes can be switched in and out as required, 'pausing' one process for a short duration while the other is being worked on, interleaving the processes in time to give the appearance of simultaneous execution.

1. **Difficulty in sharing global resources** Imagine two processes P1 and P2 both executing a block of code concurrently. Lets assume out code reads some text from a file and saves the text into a global variable. Due to concurrency we will have some form of interleaving between P1 and P2. So firstly P1 begins executing our block of code but is interrupted after reading some x character and P2 gets it time to run the CPU. P2 runs also the code and updates the global variable. P1 sets the variable however P2 updates its value after reading the rest of the text. So P1 will no longer have access to the value it set the global variable since it is lost because P2 wrote another value into it. As a result the global variable will only hold the part of text which was read from file by P2 instead of having the whole text like it should have. Usually to solve this issue we only allow only one process at a time to execute a block of code which would access global variables.

2. **Difficulty in resource allocation** When a process is accessing a shared resource (I/O device or shared data structures) no other process should be able to access this resource until the process has finished with it. This is done in order to make sure no other process can change the resource while the operation is in progress. So resources accessible to more than one process should be temporarily reserved for a single process. This makes the optimal allocation of resources difficult.

3. **Issues with debugging and programming errors** Because multiple processes update the same resource it is difficult to reproduce the contexts in which errors occur. So debugging becomes more difficult.

3.4 Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

Answer:

If a new context is already in one register set, simply set pointer of the "currently used register set" to that set.

If a new context is in memory, it needs to be loaded into one of the register sets. This will take more time.

3.5 When a process creates a newprocess using the fork() operation, which

of the following states is shared between the parent process and the child process?
a. Stack
b. Heap
c. Shared memory segments

Answer:
Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

OR

When *fork()* is called, both stack and heap will be copied into a child process. However, copying will be delayed until child process makes a change(*write*) in those memories. This is called *copy on write*.
Shared memory segements will not be copied.

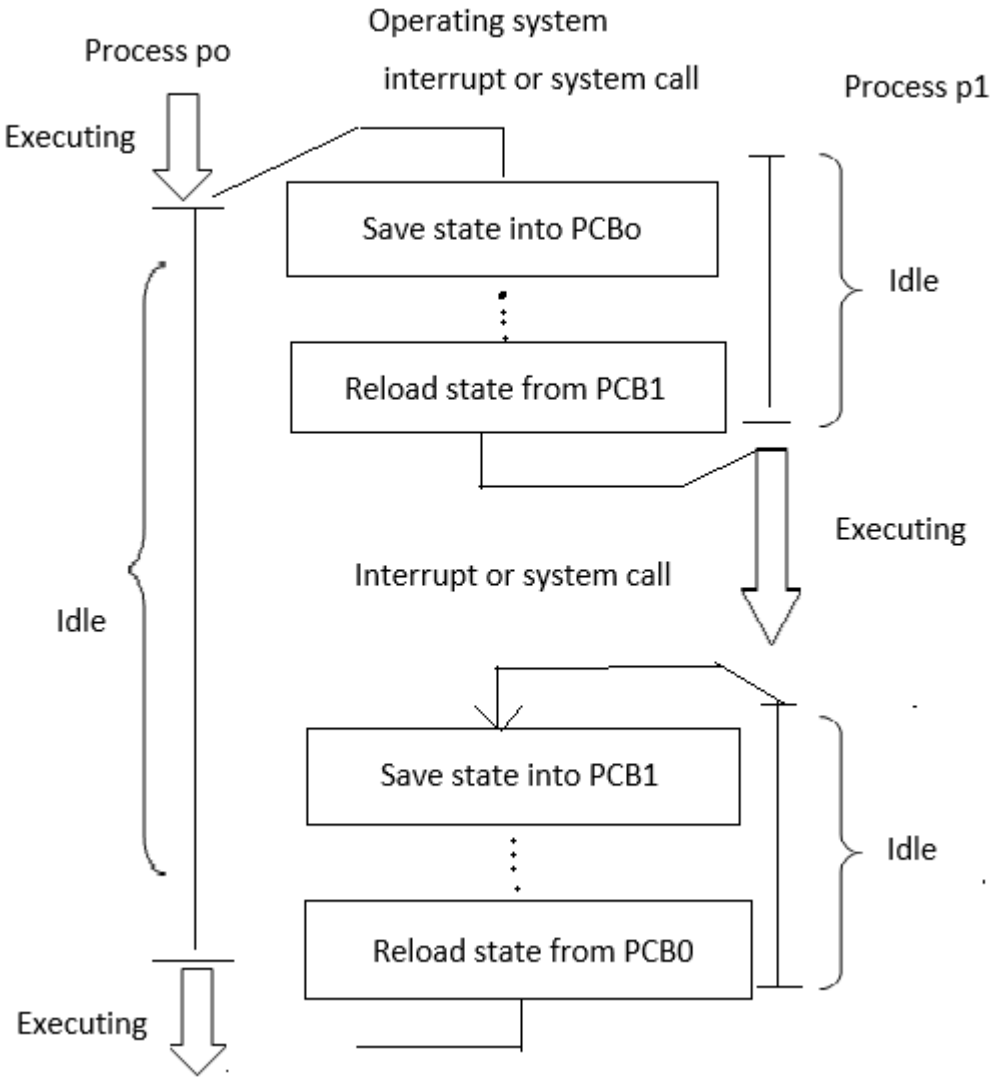
3.6 Consider the "exactly once"semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

Answer: The "exactly once" semantics ensure that a remore procedure will be executed exactly once and only once. The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages). The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock. The client will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure so the client invokes the RPC a second time, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received—and performed—by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantics for execution of RPCs?

Answer: The server should keep track in stable storage (such as a disk log) information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and a RPC message is received, the server can check whether the RPC had been previously performed and therefore guarantee "exactly once" semanctics for the execution of RPCs.

3.8 Describe the actions taken by a kernel to context-switch between processes.
Answer: In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.



- Interrupt causes the operating system to change a CPU from its current task and to run a kernel routine.
- When an interrupt occurs, the system need to save the current context of the process running on the CPU, so that it can later restore when needed.
- Switching the CPU to another process required performing a state save of the current process and state restore of different process and this task is known as context switching.
- When a context switch occur the kernel switching context of old process in d's PCB and loads the saved context of its new process scheduled to run.
- Context switch time is pare overhead, because system does not do any useful work during this time.
- It is highly depending on hardware support like if processor having large number of register so it wont need to unload old PCB as it has space enough to store all frequently used processes PCB.
- In response to clock interrupt, the OS saves the PC and user stack pointer of the current executing process and transfer control to kernel clock interrupt handler.

- The clock interrupt handler saves the rest of the register as well as other machine state such as state of floating pointer registers in the process PCB.
- The OS invoke the schedule to determine the next process to execute.
- The OS then retrieves the state of next process from cts PCB and restore the registers.
- This restore operation takes the processor back to the state in which this process was previously interrupted, executing in user mode with user mode privileges

3.10 Explain the role of the init (or systemd) process on UNIX and Linux systems in regard to process termination.

Answer :

init is the first process which started during booting of the computer system in any Unix-based operating systems. It is a daemon (a computer program that runs as a background process) and it continues running until the system is shut down. **Init** is the ancestor of all other processes (it can be a direct or indirect ancestor) and it adopts all orphaned processes.

exit() is called either directly or indirectly by a return statement in **main()** for a normal termination. When **exit()** is called and the process is terminated, this process moves to the zombie state short period until its parent process invokes a call to **wait()**. After his parent invokes the call to **wait()** both the process id and the entry in the process table are released. If his parent will not invoke **wait()** the child will become a zombie. In this case the child process will remain a zombie as long as its parent is not terminated and is still running. When the parent process terminates the without invoking **wait()** then its children will become oprhaned. In this case the **Init** process will adopt the process whose parent terminated since it is now an orphaned process. As a result **Init** will become the new parent of the zombie process. **Init** then periodically invokes **wait()** in order release the orphans PID and process-table entry.

3.11 Including the initial parent process, how many processes are created by the program shown below?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
int i;
for (i = 0; i < 4; i++)
fork();
return 0;
}
```

Answer:

These two lines of code :
for (i = 0; i <4; i++)
fork()
are equivalent to:
fork()
fork()
fork()
fork()
As a result we have one initial process forked four times.

---

first fork() After the first fork() is executed it will create a new process. So we will have now two processes. The parent and the first child. Three more forks() are left.

---

second fork() After the second fork() is executed it will create two new process. Both the parent and the child created from the first fork will be forked so we can notice two duplicate processes. As a result we will have four processes at the end of the second fork.

---

third fork() After the third fork() is executed all four processes which exist after the second fork will have a child process. So we will have now eight processes, which means four more processes are created.

---

fourth fork() After the fourth fork() is executed it will create eight new process. Each of the eight processes which already exist will have a child which means in total there will be sixteen processes. There are no more forks left.

ESULT

There are four *fork()* so in the end we will have 2^N where N = 4 so the final result will be $2^4 = 16$

3.12 Explain the circumstances under which the line of code marked printf("LINE J") in below code will be reached.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
printf("LINE J");
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
```

}
return 0;
}

Answer:

Let's look at these lines of code:
else if (pid == 0)
execlp("/bin/ls","ls",NULL);
printf("LINE J");
First the main process is forked. Then if the child is created correctly the else statement will be executed. Once the else statement is executed the code inside the child makes a call to *execlp()*. in POSIX *execlp()* is part of the exec functions family whose job is to replace the current process image with a new process image. The *exec()* command will run an executable file which is loaded into the current process space and will run it from the entry point. In our case the *execlp()* is a variant of *exec()* where the two last letters each have a meaning. **l** shows that a list is passed to the function. **p** uses the PATH environment variable to find the file named in the file argument to be executed.
So the code block above says if this is the child process then replace the child process image with a new process image which would have the effect of running a new program with the process ID of the calling process. The arguments inside *execlp* identify the location of the new process image in the hierarchical file system and the name of the executable file. Now the address space of the process is replaced with the program ls specified in the parameter. If the call to *execlp()* succeeds of course the new program will run. However if an error occurs in the call to *execlp()* the line
printf("Line J");
would be executed.

SULT

If an error occurs in the call to *execlp()* the line *printf("Line J");* would be executed.

3.13 Using the programgiven below, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid, pid1;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
pid1 = getpid();
printf("child: pid = %d",pid); /* A */
printf("child: pid1 = %d",pid1); /* B */
}
else { /* parent process */
pid1 = getpid();
```

```
printf("parent: pid = %d",pid); /* C */
printf("parent: pid1 = %d",pid1); /* D */
wait(NULL);
}
return 0;
}
```

Answer:

1

printf("child: pid = ",pid);

Firstly the **fork()**will return 0 for the child process and a number greater then 0 (in our case 2603) for the parent process. This line will print the PID which the fork will return to the process where it is executed. Since this code block is inside the child process and the **fork()** returns a zero to the newly created child process then this line will print 0.

2

printf("child: pid1 = ",pid1);

getpid() returns the process ID (PID) of the calling process. In our case the calling process is the child process. The process ID of the child process is 2603 which means the line above will print 2603.

3

printf("parent: pid = ",pid);

This line will print the PID that the **fork()** returns to the parent. Since the fork returns to the parent the PID of the child which is a value greater than 0 in our case it would print 2603.

4

printf("parent: pid1 = ,pid1);

getpid() will return the PID of the calling process. Since the calling process is the parent and its PID is 2600 the line will print 2600.

RESULT

$$A = 0, B = 2603, C = 2603, D = 2600$$

3.14 Give an example of a situation inwhich ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

Answer:

[1] A **pipe** is a technique for passing information from one process to another. **Ordinary pipes** allow two processes to communicate in standard producerconsumer fashion where the producer writes to one end of the pipe and the consumer reads from the other end. Thus they allow only one-way communication meaning they are unidirectional. An ordinary pipe cannot be accessed from outside the process that created it. Generally we use ordinary pipes when a parent process wants to communicate with its children. Once the processes have finished communicating and have terminated the ordinary pipes ceases to exist. On the other hand **named pipes** provide a much more bidirectional communication tool and don't require a parentchild relationship. Once a named pipe is established, several processes can use it for communication. From the above we can conclude that ordinary pipes work better with simple communication. An example could be a process which counts characters in a file. In this case

- The producer would write the file to the pipe
- The consumer would read the file and counts the number of characters

Next, for an example where named pipes are more suitable when several processes need to write in the same time. An example could be when several processes need to write messages to a log. A log is a simply a file that records events that occur in the OS or messages between users of a communication software. So in a log file several processes should write messages. In this case

- Several processes can o write something to the named pipe
- The messages would be read by a server from the named pipe
- The server after reading the messages writes them to a log file.

RESULT

Ordinary pipes work better with simple communication. Named pipes work better when several processes need to write messages. Ordinary pipes allow communication between parent and child processes, while named pipes permit unrelated processes to communicate.

3.15 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

Answer:

Remote Procedure Call is a form of distributed communication in a client-server system. It occurs when a process calls a procedure on a remote application. If an RPC mechanism cannot support either at most once or at least once semantics then the RPC can fail or be duplicated and executed more than once as a result of common network errors. So the RPC server cannot guarantee that a remote procedure will be invoked correctly and executed only once. If the system is unable to support these at most once or at least once semantics the it would be too dangerous to provide RPC that alter data or provide time-sensitive results.

Examples where we do not wish to enforce these semantics would be those cases where a single invocation of the RPC would result in leak of information, alteration of data or breach of security. Bank systems for example allow us to remotely perform withdrawals or payments as well as we can access our accounts information such as name or phone number through an RPC. If all we do is access our account information without altering the data we do not need to implement these semantics. Other examples could be a school account or a medical one. Inquiries into account information usually do not require at most once or at least once semantics.

ESULT

If an RPC mechanism cannot support either at most once or at least once semantics then the RPC can fail or be duplicated and executed more than once as a result of common network errors. So the RPC server cannot guarantee that a remote procedure will be invoked correctly and executed only once.

3.16 Using the program shown below, explain what the output will be at lines X and Y.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};
int main()
{
int i;
pid t pid;
pid = fork();
if (pid == 0) {
for (i = 0; i < SIZE; i++) {
nums[i] *= -i;
printf("CHILD: %d ",nums[i]); /* LINE X */
}
}
else if (pid > 0) {
wait(NULL);
for (i = 0; i < SIZE; i++)
printf("PARENT: %d ",nums[i]); /* LINE Y */
```

}
return 0;
}

Answer:

1

After the fork() the child process is a copy of it's parent and it will hold its own copy of the data which will not be affected by any change made to the parent. The parent had these initial values

int nums[SIZE] = 0,1,2,3,4;

So the child will inherit a copy of this data. Line X is executed inside the child block of code. As a result since the child has inherited the values described above after the execution of

*nums[i] *= -i;*

For each value in the array nums the program will multiply it to the negative of it's position in the array. For example the value 4 has also index 4 so the result would be 4×-4 = -16. So the final result printed at line X is

0, -1, -4, -9, -16

2

The parent had these initial values

int nums[SIZE] = 0,1,2,3,4;

On the execution of the parent's code block what the program does is it iterates through the array and prints its values.

for (i = 0; i ¡ SIZE; i++)

fprintf("PARENT: ",nums[i]);

So the final result printed at line Y is

0, 1, 2, 3, 4

RESULT

result printed on line X is **0, -1, -4, -9, -16**
result printed at line Y is **0, 1, 2, 3, 4**

3.17 What are the benefits and detriments of each of the following? Consider both the systems and the programmers' levels.

a. Symmetric and asymmetric communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

Answer:

a. **Symmetric and asymmetric communication** - A benefit of symmetric communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously; received at a point of no interest to the sender. As a result, message-passing systems often provide both forms of synchronization.

b. **Automatic and explicit buffering** -Automatic buffering provides a queuewith indefinite length; thus ensuring the senderwill never have to block while waiting to copy a message. There are no specifications how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely memory

will be wasted with explicit buffering.
c. **Send by copy and send by reference** - Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it.Abenefit of send by reference is that it allows the programmer towrite a distributed version of a centralized application. Java's RMI provides both, however passing a parameter by reference requires declaring the parameter as a remote object as well.
d. **Fixed-sized and variable-sized messages** - The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages. The number of variable-sized messages that can be held by such a buffer is unknown. Consider how Windows 2000 handles this situation: with fixed-sizedmessages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.

3.18 Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command

ps -l

The process states are shown below the S column; processes with a state of Z are zombies. The process identifier (pid) of the child process is listed in the PID column, and that of the parent is listed in the PPID column. Perhaps the easiest way to determine that the child process is indeed a zombie is to run the program that you have written in the background (using the &) and then run the command ps -l to determine whether the child is a zombie process. Because you do notwant too many zombie processes existing in the system, you will need to remove the one that you have created. The easiest way to do that is to terminate the parent process using the kill command. For example, if the pid of the parent is 4884, you would enter

kill -9 4884

Answer:

1

On Unix and Unix-like computer operating systems, a zombie is a process that has completed execution through the *exit()* system call but still has an entry in the process table. The child only becomes a zombie if it ends and the parent doesn't call *wait()* as long as the parent continues living.

2

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main()
{
    pid_t pid;
    // creating child process and storing the returned PID
    pid = fork();
    //if pid greater than 0 then the code inside if
statement is being executed by parent process
    if(pid > 0)
    {
        printf("This is the parent process");
        sleep(10);
    }
    else if (pid == 0)
    // this code is being executed by child process
    {

        printf("This is the child process");
        // terminating the child process
        exit(0);
    }
    return 0;
}
```

3

In the parent's code block described above the parent sleeps for 10 seconds while the child finishes its execution using *exit()* system call. However there is no call for *wait()* and the child becomes a zombie.

RESULT

On Unix and Unix-like computer operating systems, a zombie is a process that has completed execution through the *exit()* system call but still has an entry in the process table. The child only becomes a zombie if it ends and the parent doesn't call *wait()* as long as the parent continues living.

3.20 An operating system's **pid manager** is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. Process identifiers are discussed more fully in Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.
Use the following constants to identify the range of possible pid values:
#define MIN PID 300
#define MAX PID 5000
You may use any data structure of your choice to represent the availability

of process identifiers. One strategy is to adopt what Linux has done and use a bitmap in which a value of 0 at position *i* indicates that process id is currently in use.

Implement the following API for obtaining and releasing a pid:

- int allocate map(void)—Creates and initializes a data structure for representing pids; returns −1 if unsuccessful, 1 if successful
- int allocate pid(void)—Allocates and returns a pid; returns −1 if unable to allocate a pid (all pids are in use)
- void release pid(int pid)—Releases a pid

Answer:

```
#include<stdio.h>
#include<stdlib.h>
#define MIN_PID 300
#define MAX_PID 5000

/* We will use a bitmap in which a value of 0 at position i
indicates that
a process id of value i is available and a value of 1
indicates that the
process id is currently in use*/

using namespace std;


//Creating and initializes a data structure
//for representing pids
// returns—1 if unsuccessful, 1 if successful
struct PID_tab{
        int PID;
        bool boolean;
}*pId;

//allocating bitmap values to the data structure
int allocate_map(){

        int i;
        //void* calloc (size_t num, size_t size);
        pId=(struct PID_tab *)calloc((MAX_PID-
MIN_PID+1),sizeof(struct PID_tab));
        if(pId==NULL){
             return -1;
             }
        pId[0].PID=MIN_PID;
        pId[0].boolean=1;
        for( i=1;i<MAX_PID-MIN_PID+1;i++){
      pId[i].PID=pId[i-1].PID+1;
```

```
            pId[i].boolean=1;
        }
        return 1;
}

////allocating pid to the new process
int allocate_pid(){
   int i ;
   for( i=0;i<MAX_PID-MIN_PID+1;i++){
                if(pId[i].boolean==1){
                        pId[i].boolean=0;
                        return pId[i].PID;
                }
   }
   if(i==MAX_PID-MIN_PID+1)
        return -1;
}

//releasing pid
void release_pid(int pid){
        pId[pid-MIN_PID].boolean=1;
}

int main(){
  int pid;
  allocate_map();
  for(int i=1; i<=6; i++){
      /* This statement would be executed
       * repeatedly until the condition
       * i<=6 returns false. All the pids printed will have
different values
       */
       if((pid=allocate_pid())!=-1);
        printf("Pid= %d \n",pid);
   }
  return 0;
}
```

We want to find all the active processes PID so we can see they are unique meaning no two active processes have the same PID. So we will implement a bitmap in which a value of 0 at position i indicates that a process id of value i is available and a value of 1 indicates that the process id is currently in use.

bool boolean;

Also we have implemented a struct which is a group of data elements(can have different types and different lengths) grouped together under one name.

3.21 The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

n = { n/2, if n is even

3 × n + 1, if n is odd

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if n = 35, the sequence is

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Write a C program using the fork() system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
//we declare a variable of type integer to hold the positive
integer
//and initalize the value to zero
     int i=0;
     int status;
     //declaring the parent id
     pid_t pid;
//the exercise asks to perform necessary error checking to
ensure that a positive
//integer is passed on the command line so we will implement a
so while loop
//the statement inside the do segment will be repeated until
the number entered is positive
     do{
     //the C library function int printf(const char *format,
...) sends formatted output to stdout
          printf("Enter a positive integer: ");
     //the C library function int scanf(const char *format,
...) reads formatted input from stdin
          scanf("%d", &i);

     }while (i<=0);
    //forking the oarent function
```

```
      pid = fork();
      //fork returns a positive value to the parent
      //
      if (pid>0){
          //we don't want a zombie child
          //so we call the wait() function at the parent
         wait(&status);
      }
      //fork returns 0 to the newly created child
      // we want to generate the sequence in the child function
      else  if (pid== 0) {
         //when the algorithm is continually applied all
positive integers will eventually reach 1
         //so we want to print all values until they reach 1
         // we will use a while loop
         // we will loop until the value inputed will reach 1
           while (i!=1){
               //if inputed number is even the division with 2
will return a remainder of 0
               //so if remainder is 0 number is even, if it's 1
number is odd
               if (i%2 == 0){
                       // if number is even divide it by 2
                    i = i/2;
               }
               else if (i%2 == 1){
                // if number is odd then caculate (3*number)+1
                    i = 3 * (i) + 1;
               }
               //finally print the number
               printf("%d\n",i);
           }
           //when 1 is reached break the loop
      }

return 0;

}
```

Pseudocode

- Declare i as the number you wish to apply algorithm to, and declare the PID of the process
- Fork parent process
- If fork == 0 then its the child process
- Inside child process
  1. if the number i/ 2 has remainder of 0 then its even so calculate i/2
  2. if the number i/2 has remainder of 1 then its odd so calculate i*3+1
  3. print the number
- repeat the steps until you reach 1

3.26 Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message Hi There, the second process will return hI tHERE. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending themodified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

Answer:

1

A function switch case is created which takes a **char** array. It loops through the array and checks whether a letter is lowercase in which case it becomes uppercase or vice versa. This function will be executed by child process.

Main process first creates 2 pipes. It then forks, creating a child process. Child process will close pipe for reading, modify the message and write it to pipe. Parent process will close the pipe for writing, it will read from the pipe and output the solution.

To compile and to run the program the program via the command line use the following:

```
$ gcc exercise26.c -o exercise26.out
$ ./exercise26.out
```

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void switchCases(char* message) {
    int i = 0;
    while(message[i] != '\0') {
        if(message[i] > 64 && message[i] < 91) {
            message[i]+=32;
```

```
        }
        else if(message[i] > 96 && message[i] < 123) {
            message[i]-=32;
        }
        i++;
    }
int main(int argc, char** argv){
    int fd[2];
    char message[10] = "Hi there!";
    if (pipe(fd) == -1) {
        printf("There was an error in pipe creation. Program
will now exit.\n");
        return -1;
    }
    pid_t child;
    child = fork();
    if(child < 0) {
        printf("Forking failed.");
        return -1;
    }
    if(child == 0) {
        close(fd[0]);
        switchCases(message);
        write(fd[1],message,10);
    }
    else if(child > 0) {
        close(fd[1]);
        char read_buffer[10];
        read(fd[0],read_buffer,10);
        printf("%s",read_buffer);
    }
    return 0;
}
```

Main process first creates 2 pipes. It then forks, creating a child process. Child process will close pipe for reading, modify the message and write it to pipe. Parent process will close the pipe for writing, it will read from the pipe and output the solution.

3.27 Design a file-copying program named filecopy.c using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, ifwe invoke the program as follows:
./filecopy input.txt copy.txt
the file input.txt will be written to the pipe. The child processwill read the contents of this file and write it to the destination file copy.txt. You may write this program using either UNIX orWindows pipes.

Answer:

```
//header files for libraries
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>


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

    //declaring variable for storing the
    //reading and writing descriptor values
    //fdOne[0] will be used for reading
    //fdOne[1] will be for writing
    int fd[2];
      int fileSize;
      pid_t pid;

    char text[1024];
      char copy[1024];

    //Create two pointers for the sorce and destination files
    char* sourceFile = argv[1];
    char* destFile = argv[2];


    //creating a pipe
    //pipe() system call is used for passing information from
one process to another

    if (pipe(fd)==-1)
    {
        fprintf(stderr, "Pipe Failed" );
        return 1;
    }
    //forking the process
    pid = fork();
     if (pid < 0)
    {
        fprintf(stderr, "Fork Failed" );
        return 1;
    }
    // fork returns to parent the pid of the child
    if (pid > 0)
    {
```

```
       //declaring variables which will later be used to get
the number of bytes to read
          int sourceFileNum;
          ssize_t numBytes;
        // close the reading end of pipe, read the file and
write content to child
        close(fd[0]);
        sourceFileNum=open(sourceFile, O_RDONLY);
        numBytes=read(sourceFileNum, text,sizeof(text));
        // we need in this order for the write() system call
        //A fildescritor where to write output
        //A pointer to a buffer of at least nbytes bytes,
which will be written to the file
        //The number of bytes to write. If smaller than the
provided buffer the output is truncated.
        write(fd[1],text, numBytes);
        //close the writing end of pipe
        close(fd[1]);
    // the child process
    }else if (pid == 0){

      int destDesc;
        // close writting end of pipe
        close(fd[1]);
        ssize_t numBytesCh;
        numBytesCh =read(fd[0], copy,sizeof(copy));
        // close reading end of pipe
        close(fd[0]);
        // Open a file for writing and write content to
newfile descriptor
        destDesc=open(destFile, O_CREAT | O_WRONLY);
        // we need in this order for the write() system call
        //A fildescritor where to write output
        //A pointer to a buffer of at least nbytes bytes,
which will be written to the file
        //The number of bytes to write. If smaller than the
provided buffer the output is truncated.
        write(destDesc, copy, numBytesCh);
    }

    return 0;
}
```

In the command line in the UNIX environment create a file with some text in it input.txt and another empty file copy.txt. After you execute the program the child process will read the contents of input.txt file and write it to the destination file copy.txt.