**P. 124,125**

**Q2**. Can UNIX exec return an error? Why or why not? Note: You can answer this question by looking at the manual page for exec, but before you do that, think about what the exec system call does. If you were designing this call, would you need to allow it to return an error?

Yes, the exec() function only returns if it has an error, then it will return with a -1. If it does not get an error then it will not return and will complete the function. It needs to return when there’s an error to let the user/calling program know that there has been an issue that needs to be resolved. Some of the reasons that the exec() function could fail is that the program that its trying to call doesn’t exist, of that it cannot access the program, or that it doesn’t have the correct permissions to be able to run the program. Yes, if I was creating this function, I would most definitely allow it return errors to help troubleshooting the issue if the function was not able to complete successfully. I suppose that you could have it always return a 1 when the function executes successfully and just exit upon an error. That would make trouble shooting very difficult though.

**Q10**. What is the output of the following programs? (Please try to solve the problem without compiling and running the programs.)

|  |  |
| --- | --- |
| Program 1: main(){  int val = 5;  int pid;     if(pid = fork())         wait(pid);     val++;     printf("%d\n", val);     return val; } | Program 2: main(){     int val = 5;  int pid;     if(pid = fork())         wait(pid);     else         exit(val);     val++;     printf("%d\n", val);     return val; } |

|  |  |
| --- | --- |
| Program 1 The parent process creates a child and then waits for the child to exit because of the wait(). The child executes and prints out val which is 6 and the after the val++ the child returns the val to the parent and satisfies the wait(). The parent then prints out the final val which is now 7. | Program 2 is very similar and when we look at it. We see that the parent waits for the child, and the child immediately exits and returns the final val of 6 to the parent and it gets printed and returned. |

**P.179**

**Q5**. Write a program that uses threads to perform a parallel matrix multiply. To multiply two matrices, C = A \* B, the result entry C (i,j) is computed by taking the dot product of the ith row of A and the jth column of B: c(I,j) = sigma N-1, k=0 A(I,j) B(I,j). We can divide the work by creating one thread to compute each value (or each row) in C, and then executing those threads on different processors in parallel, as shown in Figure 4.19.

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  // maximum size of matrix  #define MAX 4  #define MAX\_THREAD 4  int matA[MAX][MAX];  int matB[MAX][MAX];  int matC[MAX][MAX];  int step\_i = 0;  void\* multi(void\* arg)  {  int core = step\_i++;  for (int i = core \* MAX / 4; i < (core + 1) \* MAX / 4; i++)  for (int j = 0; j < MAX; j++)  for (int k = 0; k < MAX; k++)  matC[i][j] += matA[i][k] \* matB[k][j];  }  int main()  {  for (int i = 0; i < MAX; i++) {  for (int j = 0; j < MAX; j++) {  matA[i][j] = rand() % 10;  matB[i][j] = rand() % 10;  }  }  cout << endl  << "Matrix A" << endl;  for (int i = 0; i < MAX; i++) {  for (int j = 0; j < MAX; j++)  cout << matA[i][j] << " ";  cout << endl;  }  cout << endl  << "Matrix B" << endl;  for (int i = 0; i < MAX; i++) {  for (int j = 0; j < MAX; j++)  cout << matB[i][j] << " ";  cout << endl;  }  pthread\_t threads[MAX\_THREAD];  for (int i = 0; i < MAX\_THREAD; i++) {  int\* p;  pthread\_create(&threads[i], NULL, multi, (void\*)(p));  }  for (int i = 0; i < MAX\_THREAD; i++)  pthread\_join(threads[i], NULL);    cout << endl  << "Multiplication of A and B" << endl;  for (int i = 0; i < MAX; i++) {  for (int j = 0; j < MAX; j++)  cout << matC[i][j] << " ";  cout << endl;  }  return 0;  } |
|  |