Processes	**LABORATORY**	**FOUR**
OBJECTIVES		
1. Unix Time 2. Programming with Processes		

Unix Time and Y2K Problem

In Unix or Linux, the *time* is simply a *counter* (of data type **`time_t`**) that counts the number of seconds from a specific date, namely, **`1 January 1970, 00:00:00`** and a value of 86399 means 1 January 1970, 23:59:59. The integer counter is of size 32 bits. In each day, there are 24 hours, i.e. 86400 seconds. Each year is about 86400 * 365.2422 = 31556926 seconds. The size of a 32-bit integer can count up to 2147483647, i.e. a little more than 68 years. So the Unix time counter will overflow by year 2038. Remember the **Y2K** problem that people are using two digits (bytes) for the year instead of four digits to save storage and by the year 2000, the date will wrap around to 1900. This **2038 problem** is the **Unix Y2K bug** or **Unix Millennium bug.** Many Unix programs would not function correctly by 19 January 2038 or when computing a date beyond that day. How many of you have heard of this **Unix Y2K bug**?

Programmers are already advised to write their programs with checking for the invalid entry for the date data type **`time_t`** to avoid this **2038 problem** (**Y2K38 problem**) on Unix and Linux. Newer architectures based on 64-bit integers would no longer suffer from this problem by replacing the **`time_t`** data type. However, people need time to switch to new architectures and need to watch out for legacy codes and programs that rely on this 32-bit time representation. Could you *estimate* ***when*** will an overflow to this new 64-bit time data type occur? Remember that we still need to handle the **Y10K problem** in another 8000 years, after solving the **Y2K problem**.

Try to study **lab4A.c** for the calendar logic, and figure out how this program can be executed. You should also try to learn the skill of *table-lookup* for fast result determination. This is a very useful programming skill. As a *simple exercise*, you could *extend* the checking logic for leap years for year 1900, 2000 and 2100. When the year ends in "00", it must be divisible by 400 to qualify as a leap year. So, years 1900 and 2100 are ***not*** leap years, but year 2000 is a leap year.

```
// lab 4A
// try to find out how this calendar-related program can be executed
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  char *name[] = {"Jan", "Feb", "Mar", "Apr", "May", "June",
                  "July", "Aug", "Sept", "Oct", "Nov", "Dec"};
  int yy, mm, dd, numDay;
  int i;

  yy = atoi(argv[1]); numDay = atoi(argv[2]); // atoi returns an integer
  if (numDay <= 0 || numDay > 366 || (numDay == 366 && yy % 4 != 0)) {
     printf("Sorry: wrong number of days\n");
     exit(1);
  }
  if (yy % 4 == 0) month[1] = 29;
  for (i = 0; i < 12; i++) {
      if (numDay <= month[i]) break;
      numDay = numDay - month[i];
  }
  mm = i; dd = numDay;

  printf("Date is %d %s %4d\n", dd, name[mm], yy);
}
```

Programming with Processes

In Unix/Linux the way to create a process is to use the **`fork()`** system call. It will create a child process as an *exact copy* of the parent, including the value of the program counter. However, once **`fork()`** is executed, the two copies of parent and child will be *separated* and the variables are ***not shared***. Note that the action of copying only occurs on demand but logically, we could assume that it occurs when **`fork()`** is completed. To get the id of a process, we could use the system call **`getpid()`**. The **`sleep()`** system call allows a process to stop for the specific number of seconds. The **`exit()`** system call allows the process to terminate successfully.

Since the two copies of parent and child are separated after **`fork()`**, how could a child know who is the parent? A child can get the id of the parent by **`getppid()`**. Try to *observe the variable values* produced by parent and child processes and *draw a picture to illustrate* the changes in values in **lab4B.c**. Note that there is no guarantee that the parent after **`fork()`** is executed before or after the child. The actual execution order depends on the CPU scheduling mechanism. Vary the **`sleep()`** parameters and observe the outputs. You should be able to generate different outputs by properly changing the waiting time.

```
// lab 4B
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  childid, myid, parentid, cid;
  int  val;

  val = 1;
  myid = getpid();
  printf("My id is %d, value is %d\n",myid,val);
  childid = fork();
  if (childid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (childid == 0) { // child process
     myid = getpid(); parentid = getppid();
     printf("Child: My parentid is %d, childid is %d\n",parentid,childid);
     printf("Child: My id is %d, value is %d\n",myid,val);
     val = 12;
     printf("Child: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Child: My id is %d, value is %d\n",myid,val);
     val = 13;
     printf("Child: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Child: My id is %d, value is %d\n",myid,val);
     printf("Child: I %d completed\n",myid);
     exit(0);
  } else { // parent process
     printf("Parent: My childid is %d\n",childid);
     printf("Parent: My id is %d, value is %d\n",myid,val);
     val = 2;
     printf("Parent: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Parent: My id is %d, value is %d\n",myid,val);
     val = 3;
     printf("Parent: My id is %d, value is %d\n",myid,val);
     sleep(4);
     printf("Parent: My id is %d, value is %d\n",myid,val);
     cid = wait(NULL);
     printf("Parent: Child %d collected\n",cid);
     exit(0);
  }
}
```

Again, since the child is an *exact copy* of the parent, all values stored in the variables before **`fork()`** are *exactly the same*. All values stored in the argument list and all other variables are directly *inherited* by the child. This serves as a simple mechanism of passing information from parent to child, in case the information is known before the creation of the child process via **`fork()`**.

Here is a program to compute the GPA of subjects of different levels, still based on the old GPA system. Would there be any difference if we create the child process in the first statement, i.e. move **`fork()`** to the *beginning*? In other words, move the line **`childid = fork();`** to the first line, before **`num_subj = (argc-1)/2;`** and **`printf("There are %d subjects\n",num_subj);`** Try to answer the question *before* you actually change the program and run it to see the actual outcome.

```
// lab 4C
#include <stdio.h>
#include <stdlib.h>

float computeGP(char grade[])
{
  float gp;

  switch (grade[0]) {
     case 'A': gp = 4.0; break;
     case 'B': gp = 3.0; break;
     case 'C': gp = 2.0; break;
     case 'D': gp = 1.0; break;
     case 'F': gp = 0.0; break;
     default: printf("Wrong grade %s\n",grade);
          return -1.0; // use a negative number to indicate an error
  }
  if (grade[1] == '+') gp = gp + 0.3;
  if (grade[1] == '-') gp = gp - 0.3;
  return gp;
}

int main(int argc, char *argv[])
{
  float gp, sum_gp = 0.0;
  int childid, cid;
  int num_subj, sub_code, count;
  int i;

  num_subj = (argc-1)/2;
  printf("There are %d subjects\n",num_subj);

  childid = fork();
  if (childid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (childid == 0) { // child process: handle level 3 and level 4 subjects
     count = 0;
     for (i = 1; i <= num_subj; i++) {
          sub_code = atoi(argv[i*2-1]); // convert into integer
          if (sub_code >= 3000) {
             gp = computeGP(argv[i*2]);
             if (gp >= 0) { count++; sum_gp += gp; }
          }
     }
     printf("Child: ");
     printf("Your GPA for %d level 3/4 subjects is%5.2f\n",count,sum_gp/count);
     exit(0);
  } else { // parent process: handle level 1 and level 2 subjects
     count = 0;
     for (i = 1; i <= num_subj; i++) {
          sub_code = atoi(argv[i*2-1]); // convert into integer
          if (sub_code < 3000) {
             gp = computeGP(argv[i*2]);
             if (gp >= 0) { count++; sum_gp += gp; }
          }
     }
     printf("Parent: ");
     printf("Your GPA for %d level 1/2 subjects is%5.2f\n",count,sum_gp/count);
     cid = wait(NULL);
     printf("Parent: Child %d collected\n",cid);
     exit(0);
  }
}
```

You could modify the program with the technique in **lab4A.c** to print out English words rather than just numbers (assuming an upper bound on the number of subjects). A simple table lookup is often more effective than using a long list of **`switch/case`** statements. You could further compute the *weighted GPA* once you separate them into levels, i.e. level 1 and 2 subjects have a weight of 2 and level 3 and 4 subjects have a weight of 3. Do you find a bug there in **lab4C.c**?

```
lab4C 1011 B 3121 C+ 2011 C+ 4122 B- 3911 B+
There are five subjects
Child: Your GPA for three level 3/4 subjects is 2.77
Parent: Your GPA for two level 1/2 subjects is 2.65
Parent: Child 13579 collected
```

A simple way for the child process to return some brief information (just a byte from 0 to 255) back to the parent is to make use of the **`exit()`** system call, riding on the Unix/Linux mechanism of exit status reporting. By convention, any user or system program would return **`0`** if the program is executed successfully and non-zero otherwise. However, as a C programmer, it may be convenient to make use of this exit status for a child to return some other agreed-upon value back to the parent. Note that this is more a workaround approach to get return values, than a formal communication mechanism between parent and child.

An example for the parent to get back what the child would be returning via the **`exit()`** system call is shown. Instead of passing in **`NULL`** as the argument, you pass in *a pointer to an integer* to the **`wait()`** system call. Then the integer will contain the exit status in its *least significant byte*. Since **apollo** or **apollo2** is a little Endian machine in terms of hardware, you can extract this return value by dividing it by 256 and taking the quotient. That is not a good approach, since the program is *not portable* to a big Endian machine. A better way is to use the macro **`WEXITSTATUS()`** to extract the exit status as a *byte*, which works regardless of the machine type. Try changing the exit status value in the child process and see how the changed value is returned from the child to the parent.

```
// lab 4D
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  childid, myid, parentid, cid;
  int  ret, cret;

  myid = getpid();
  childid = fork();
  if (childid < 0) { // error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (childid == 0) { // child process
     myid = getpid();
     parentid = getppid();
     printf("Child: My id is %d, my parentid is %d\n",myid,parentid);
     // do some calculation here
     ret = 3; // just an example here
     exit(ret);
  } else { // parent process
     printf("Parent: My childid is %d\n",childid);
     cid = wait(&cret);
     printf("Parent: Child %d collected\n",cid);
     printf("Parent: Child returning value %d vs %d\n",cret,WEXITSTATUS(cret));
     exit(0);
  }
}
```

Orphan Processes (Optional)

In Unix, if a parent process terminates before a child process terminates, the child process would become an orphan will be adopted by a new parent. Orphan processes will be adopted by process **init** with pid 1, which becomes the *new parent*. This approach is also used by Linux. Run the following program in *background* (**`lab4E o &`**) on **apollo2** and on a **Mac**. Type **`ps -lf`** and you will see that the orphan child process has been ***adopted*** by the process called **init** with pid **1** (look at the **PPID** column of information). This process with pid 1 is the **systemd** (**system daemon**) process on CentOS Linux. If a child terminates before a parent, it would become a zombie until waited or collected by its parent via **`wait()`** system call before the parent terminates. Run the program in *background* again (**`lab4E z &`**). Type **`ps -lf`** and you will see a *zombie process*. A zombie process is identified by a "**Z**" state (on the "**S**" or "**STAT**" column). Alternatively, you may use multiple windows to run and check, without putting the processes to the background.

```
// lab 4E : o means orphan and z means zombie
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int childid;

  childid = fork();
  if (childid < 0) { // an error occurred
     printf("Fork Failed\n");
     exit(1);
  } else if (childid == 0) { // child process
     if (argv[1][0] == 'o')
        sleep(30); // child still executing when parent terminates
     printf("Child completed\n");
     exit(0);
  } else { // parent process
     // parent does not wait for child
     if (argv[1][0] == 'z')
         sleep(30); // child completes before parent
     printf("Parent completed\n");
     exit(0);
  }
}
```

Laboratory Exercise

Making use of **lab4C.c** for child process generation, create a program that will *generate a **separate** child process* to carry out *strength evaluation* of its share of a deck of cards dealt in the **Bridge Game**. There are ***n*** players in this game (***n*** = 4). The parent is just like a *server* that accepts an input and passes it to a child process to do the evaluation. Remember that a child will know *everything* about the parent before **`fork`** is executed, so there is no need for the parent to pass in any argument to the child (and this simplifies the program design). The child should be able *to extract its own part* of the input data knowing its position from the argument list or from any stored array before **`fork`**.

When the program is executed, each argument in the input refers to a card in a deck. There are four suits in a card, namely, *Spade* (♠), *Heart* (♥), *Diamond* (♦), and *Club* (♣). There are also thirteen ranks, namely, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **10**, **J**, **Q**, **K**, and **A**, where **A** is the largest card and **2** is the smallest card. To simplify printing, the suits are called **S**, **H**, **C** and **D** respectively, and the card ranked **10** is represented as **T**.

The cards are dealt to each child in turn, in a round-robin manner. The first child will get cards 1, 5, 9, 13 and so on; the second child will get cards 2, 6, 10, 14 and so on. Each child should print out the cards that it receives, called its *hand*. It then evaluates the strength of its hand and print out the result.

The strength of a hand is determined by the *high card points*. High card points are awarded to *honor cards* (**J**, **Q**, **K**, and **A**). Each **A** is worth 4 points. Each **K** is worth 3 points. Each **Q** is worth 2 points. Each **J** is worth 1 point. All other cards are worth nothing. That is of course a rather coarse evaluation, but is often sufficient for beginners in deciding the bidding strategy. Besides the high card points, the distributional feature of the hand is also an important factor affecting the value of the hand. In **Bridge Game**, a **`void`** (a suit without any card) is highly valuable (that allows a trump card to be played); a **`singleton`** (a suit with just one card) is quite valuable; a **`doubleton`** (a suit with two cards) is of some value. A long suit (five cards or more) is also of a good value. On the contrary, a **`singleton`** honor that is not **A** may be dropped by the opponent and is worth less than it should have been. So adjustment to high card points would be needed for a fairer evaluation.

If there is a five-card suit, 1 point will be added. If there is a six-card suit, 2 points will be added. If there is a seven-card suit (or more), 3 points will be added. A **`void`** is worth 3 points. A **`singleton`** is worth 2 points. A **`doubleton`** is worth 1 point. Any **`singleton`** honor card is worth 1 point ***less*** than its original value (including **A** since it is less flexible in materializing its value during the play).

Your program should be able to evaluate the quality of a hand for each child. The evaluation is carried out by the child processes and the parent is not doing any real work. You can assume that there are no errors in the user input and there are no duplicated cards. Do not forget to ***wait*** for the child to complete to avoid *zombie processes*.

Please provide ***appropriate comments*** and check to make sure that your program can be ***compiled*** and ***execute*** properly under the Linux (**apollo** or **apollo2**) environment before submission. Note that you have to let all children *execute together* and *must not* control the output order from different children. The final results produced by all the children may be in any mixed order and you *do not need* to do anything to change the output order. Just let the nature play the output game for you.

Sample executions:

```
bridge S5 C7 CQ SK HT HA H5 D4 DK D6  S3 C8 S4 S7 HQ DJ H7 SQ C5 D9 S6 HK D8 C4 S8
H6 H8 HJ D2 H4 CT  SA H3 C3 D5 ST H2 CK  C2 D7 DT S2 H9 D3 DA C9 CJ DQ S9 SJ C6 CA

bridge H6 SQ D9 S5 C5 S8 C3 DT C7 SA C6 S9 C8 H3 S4 HT S2 DQ SK DK S3 C9 H8 S6 HQ H7
S7 CK D4 DA HK HA D6 H4 D7 D8 C4 DJ D2 H5 CJ CA CT HJ C2 CQ H9 H2 SJ ST D5 D3
```

Sample outputs:

```
Child 1, pid 23460: S5 HT DK S4 H7 S6 S8 D2 H3 H2 DT DA S9
Child 2, pid 23461: C7 HA D6 S7 SQ HK H6 H4 C3 CK S2 C9 SJ
Child 1, pid 23460: <S9 S8 S6 S5 S4> <HT H7 H3 H2> <DA DK DT D2> <>
Child 3, pid 23462: CQ H5 S3 HQ C5 D8 H8 CT D5 C2 H9 CJ C6
Child 2, pid 23461: <SQ SJ S7 S2> <HA HK H6 H4> <CK C9 C7 C3> <D6>
```

```
Child 4, pid 23463: SK D4 C8 DJ D9 C4 HJ SA ST D7 D3 DQ CA
Child 2, pid 23461: 13 points, 15 adjusted points
Child 4, pid 23463: <DQ DJ D9 D7 D4 D3> <SA SK ST> <CA C8 C4> <HJ>
Child 3, pid 23462: <CQ CJ CT C6 C5 C2> <HQ H9 H8 H5> <D8 D5> <S3>
Child 3, pid 23462: 5 points, 10 adjusted points
Child 1, pid 23460: 7 points, 11 adjusted points
Child 4, pid 23463: 15 points, 18 adjusted points
```

```
Child 1, pid 23466: H6 C5 C7 C8 S2 S3 HQ D4 D6 C4 CJ C2 SJ
Child 2, pid 23467: SQ S8 SA H3 DQ C9 H7 DA H4 DJ CA CQ ST
Child 4, pid 23469: S5 DT S9 HT DK S6 CK HA D8 H5 HJ H2 D3
Child 4, pid 23469: <HA HJ HT H5 H2> <DK DT D8 D3> <S9 S6 S5> <CK>
Child 2, pid 23467: <SA SQ ST S8> <H7 H4 H3> <DA DQ DJ> <CA CQ C9>
Child 4, pid 23469: 11 points, 13 adjusted points
Child 3, pid 23468: D9 C3 C6 S4 SK H8 S7 HK D7 D2 CT H9 D5
Child 2, pid 23467: 19 points, 19 adjusted points
Child 3, pid 23468: <D9 D7 D5 D2> <SK S7 S4> <HK H9 H8> <CT C6 C3>
Child 1, pid 23466: <CJ C8 C7 C5 C4 C2> <SJ S3 S2> <HQ H6> <D6 D4>
Child 1, pid 23466: 4 points, 8 adjusted points
Child 3, pid 23468: 6 points, 6 adjusted points
```

Level 1 requirement: the child processes are created and capable of printing out the cards received.

Level 2 requirement: the child processes can separate the cards into four suits and print out the number of high card points correctly.

Level 3 requirement: the child processes can print out the cards sorted according to distribution and compute and print the correct points and adjusted points.

Bonus level: the child processes can handle the dealing of two more joker cards, **JJ** and **jj**, which are considered wild cards in the game to replace a non-duplicate card in the hand. Determine which card that a joker would represent to produce a best hand with the highest adjusted points. When joker cards are available, the final two cards in the deck will be discarded (since now the deck contains 54 cards and each hand contains exactly 13 cards).

Name your program **queue.c** and **submit it via BlackBoard on or before 10 March 2024 (Sunday)**.