

Processes	LABORATORY	THREE
<b>OBJECTIVES</b>		
1. More Unix and Linux 2. Process Management 3. Simple Programming with Processes		

### A Little More Unix/Linux

In Unix/Linux, you could know more about the hardware platform that you are using with the `uname` command. Try out `uname -a` on your **Mac** and on **apollo2**. You can know the name of your machine by `hostname`. Connecting these together with `whoami` and `pwd`, you can change the command prompt easily. For example, to make the prompt

```
Mickey@apollo2:
```

you could type `set prompt="Mickey@`hostname` : "`

To make the prompt

```
/home/11234567d (11234567d)-
```

you could type `set prompt="`pwd` (`whoami`)- "`

To make the prompt

```
Linux@apollo2 (30):
```

where 30 is current command number, type `set prompt="`uname`@`hostname` (\!):"`

### 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** after 8000 years, after solving the **Y2K problem**.

Try to study **lab3A.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 may 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 3 A : 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);
}

```

## Process Management

In Unix/Linux, one or more processes will be created when executing programs using the shell. The set of processes could be viewed by means of the **ps** command. For example, you could type command **ps -elf** to see a list of processes. Each process is identified by a process id. You may be able to see a *zombie process* on **apollo2**. A zombie process is identified by a “Z” state (on the “S” or “STAT” column). Most processes should be in “S” state (*suspended*). The process executing **ps** is itself in “R” state (*running*). You may monitor the list of processes currently executing in the machine, by the command **top**. You should be seeing many more processes on **apollo2**, inclusive of other students’.

If you find that a particular process may have been caught into an infinite loop, you could attempt to terminate that process by *killing* it. You will use the **kill** command, by supplying the process ID, which can be seen when you type the command **ps -elf** for the list of processes, e.g., to terminate process 12345, type **kill 12345** and if you still cannot kill it, use the stronger option **kill -9 12345**

The invocation of a program is called a *job* or a *task* in Unix/Linux. A job normally creates one *process*, but could also create multiple processes to complete the mission. For example, the commonly used *pipe* or “|” operator in shell generates multiple processes. For example, type **ls -l | more** and you can use **ps** to check that there are two processes created, one for **ls -l** and the other for **more**. The **ls** and **more** commands are executed together for a job. In this case, you have to use another shell (e.g. use another *ssh client*) to check because you could not type the **ps** command when the **ls** and **more** commands are still being executed. We could try a *sequence of commands* for a job like **ps; pwd; cal 2011**. Processes belonging to the same job form a *process group*.

You could use multiple *ssh clients* when you are using the computers in the laboratory so that each program you want to execute can be run on different windows (*ssh* or *putty* sessions). However, if you are connecting from home, you may have only one terminal or only one connection to Unix/Linux. Then could we do multiple jobs together? Luckily, the answer is *yes*, and it works with C-shell and Bourne-again shell. At any moment, there is only one process group derived from one job that is connected to the keyboard for input, but there could be multiple jobs not connected to the keyboard running at the same time. We say that these multiple jobs, except for the one connected to the keyboard, are being run in *background*. All these multiple background processes could still produce outputs to the screen, and all the outputs from all the jobs will be *intermixed* together. To put a job to background, use the “&”

operator. Try the following long printing program **lab3B.c** and you could see an intermixed output. Type **ps -lf** to see your processes.

```
cc lab3B.c -o lab3B
./lab3B first 50 &
./lab3B second 60 &
./lab3B third 40 &
```

Repeat the above execution of the three processes on three different windows (terminals). Here we do not need to use “&” operator. All processes can receive inputs from the keyboard and produce outputs to the appropriate windows. The fact is that you may not always have the luxury of multiple windows.

Do you find that the lines are intermixed together to make it very difficult to type in the new command when a single window is used? To relieve the problem, you could create a script to contain all the commands. For example, you could create a (script) file called **allLab3** to contain the following lines:

```
./lab3B first 50 &
./lab3B second 60 &
./lab3B third 40 &
```

You should compile the program first and then type **chmod 700 allLab3** to make the script file **allLab3** executable. You can then simply execute the jobs by typing **allLab3** and you will see a better picture. The effect is like typing the three lines on the keyboard by yourself, like a *batch file*.

Command	Usage	Description
<b>bg</b>	<b>bg</b>	Move a foreground job to background
<b>fg</b>	<b>fg %2</b>	Move a background job to foreground
<b>jobs</b>	<b>jobs</b>	Show the list of background jobs
<b>kill</b>	<b>kill %2</b>	Terminate a background job
	<b>kill 4321</b>	Terminate a process
	<b>kill -9 4321</b>	Terminate a process
<b>time</b>	<b>time a.out</b>	Show the total execution time of a program and its share of CPU time
<b>uptime</b>	<b>uptime</b>	Show the current workload of the computer, e.g. number of users
<b>top</b>	<b>top</b>	Show the CPU and resource utilization of the computer with a list of active processes

## 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()**.

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 **lab3C.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.

```
/* lab 3 C */
#include <stdio.h>
#include <stdlib.h>

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

    val = 1;
    myid = getpid();
```

```

printf("My id is %d, value is %d\n",myid,val);
pid = fork();
if (pid < 0) { /* error occurred */
    printf("Fork Failed\n");
    exit(1);
} else if (pid == 0) { /* child process */
    myid = getpid(); parentid = getppid();
    printf("Child: My id is %d, my parent pid is %d\n",myid,parentid);
    printf("Child: Pid is %d, value is %d\n",pid,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: Child %d completed\n",myid);
    exit(0);
} else { /* parent process */
    printf("Parent: My child id is %d\n",pid);
    printf("Parent: Pid is %d, value is %d\n",pid,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);
}
}

```

**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* (**lab3D 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). 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 (**lab3D z &**). Type **ps -lf** and you will see a *zombie process*. Alternatively, you may use multiple windows to run and check, without putting the processes to the background.

```

/* lab 3 D : o means orphan and z means zombie */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int pid;

    pid = fork();
    if (pid < 0) { /* an error occurred */
        printf("Fork Failed\n");
        exit(1);
    } else if (pid == 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 **lab3C.c** for child process generation, create a program that will *generate a separate child process* to carry out the *counting* of its share of a deck of card dealt in the **Big 2 Game**. There can be at most 10 players and at least 3 players. The parent is just like a *server* that accepts an input and passes it to a child process to do the counting. 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 and also perform result printing for the parent after counting.

When the program is run, the first argument is the number of child processes to be created. Each remaining input refers to a card in a deck. There are four suits in a card, namely, *Spade* (♠), *Heart* (♥), *Club* (♣), and *Diamond* (♦). There are also thirteen ranks, namely, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A and 2, where 2 is the largest 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. If there are 3 children, the first child will get cards 1, 4, 7, 10 and so on; the second child will get cards 2, 5, 8, 11 and so on; the third child will get cards 3, 6, 9, 12 and so on. Each child should sort the cards that it receives and print them out. It then identifies all 5-card patterns in his/her set of cards. The highest 5-card pattern is **straight flush**, consisting of five cards of consecutive ranks in the same suit, e.g., `<SK,SQ,SJ,ST,S9>`, `<H6,H5,H4,H3,H2>`. This is followed by **four of a kind**. The next pattern is **full house**, both of which will be skipped for simplicity. This is followed by **flush** (*flower*), consisting of five cards of the same suit but not in consecutive ranks, e.g., `<DT,D8,D5,D4,D3>`. The final pattern is **straight** (*snake*), consisting of five cards of consecutive ranks, but heterogeneous suit, e.g., `<S7,H6,D5,H4,C3>`. It is really not simple to list all possible patterns (recall your COMP 210 knowledge on counting). If you can find 7 consecutive cards in a **straight flush** such as `<SA,SK,SQ,SJ,ST,S9,S8>`, there are three possible **straight flushes**: `<SA,SK,SQ,SJ,ST>`, `<SK,SQ,SJ,ST,S9>`, `<SQ,SJ,ST,S9,S8>`. If you can find 7 cards in the same suit, there will be 21 possible choices for **flushes**. Counting the number of **straight** is even more complicated.

Let us simplify the requirement to avoid complicated counting: print the *longest straight flush*, the *longest flush* and the *longest straight*, ignoring **four of a kind** and **full house**. If there are more than one straight flushes or straights of equal length, print the *largest one*. However, if there are more than one flushes of equal number of cards, print the one with the *highest suit*. For multiple choices of cards in the same rank for a **straight**, print the card with the *highest suit*. If there is no 5-card pattern, say so. Note that `<J,Q,K,A,2>` is *not* considered as a **straight**. Although `<A,2,3,4,5>` is considered as a **straight**, it is considered to be *smaller* than the **straight** `<7,6,5,4,3>` since in order to make it a **straight**, both card **2** and card **A** need to be treated with ranking below the card **3** as very special arrangement. It is even smaller than `<2,3,4,5,6>` for the same reason. 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 **run** properly under the Linux (**apollo2**) environment before submission. Note that you have to let all children *run together* and *must not* control the output order from different children. The final results produced by all the children can 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:

```
deal 4 D4 HK CA DA SK HQ CK DK SQ HJ CQ DQ SJ HT CJ DJ ST H9 CT HA S9 C9 H8 D9 S7
H6 D8 C8 S2 H5 C6 C3 S6 H4 C5 D6 DT H3 C4 D5 S4 H2 H7 C7 S3 S8 C2 D3 S5 D7
SA D2
```

```
deal 3 D4 HK CA DA SK HQ CK DK SQ HJ CQ DQ SJ HT CJ DJ ST H9 CT HA S9 C9 H8 D9 S7 H6
D8 C8 S2 H5 C6 C3 S6 H4 C5 D6 DT H3 C4 D5 S4 H2 H7 C7 S3 S8 C2 D3 S5 D7 SA D2
```

Sample outputs:

```

Child 1, pid 23456: I have 13 cards
Child 2, pid 23457: I have 13 cards
Child 4, pid 23459: I have 13 cards
Child 3, pid 23458: I have 13 cards
Child 1, pid 23456: S2 SK SQ SJ ST S9 S7 S6 S5 S4 S3 DT D4
Child 4, pid 23459: HA C8 C7 C3 D2 DA DK DQ DJ D9 D6 D5 D3
Child 2, pid 23457: S8 H2 HK HQ HJ HT H9 H6 H5 H4 H3 C9 D7
Child 3, pid 23458: SA H8 H7 C2 CA CK CQ CJ CT C6 C5 C4 D8
Child 1, pid 23456: straight flush length 6 <S7,S6,S5,S4,S3,S2>
Child 2, pid 23457: straight flush length 5 <HK,HQ,HJ,HT,H9>
Child 4, pid 23459: no straight flush
Child 2, pid 23457: flush length 10 <H2,HK,HQ,HJ,HT,H9,H6,H5,H4,H3>
Child 4, pid 23459: flush length 9 <D2,DA,DK,DQ,DJ,D9,D6,D5,D3>
Child 2, pid 23457: straight length 12 <HK,HQ,HJ,HT,H9,S8,D7,H6,H5,H4,H3,H2>
Child 3, pid 23458: straight flush length 5 <CA,CK,CQ,CJ,CT>
Child 1, pid 23456: flush length 11 <S2,SK,SQ,SJ,ST,S9,S7,S6,S5,S4,S3>
Child 1, pid 23456: straight length 6 <S7,S6,S5,S4,S3,S2>
Child 3, pid 23458: flush length 9 <C2,CA,CK,CQ,CJ,CT,C6,C5,C4>
Child 4, pid 23459: straight length 5 <D9,C8,C7,D6,D5>
Child 3, pid 23458: straight length 5 <SA,CK,CQ,CJ,CT>

Child 1, pid 23461: I have 18 cards
Child 2, pid 23462: I have 17 cards
Child 3, pid 23463: I have 17 cards
Child 2, pid 23462: S2 SK ST S4 HA HK HT H8 H6 H3 C2 CQ C7 C5 C3 DK D7
Child 1, pid 23461: SJ S8 S7 S5 HJ H7 H4 CK CT C9 C8 C6 D2 DA DJ DT D5 D4
Child 3, pid 23463: SA SQ S9 S6 S3 H2 HQ H9 H5 CA CJ C4 DQ D9 D8 D6 D3
Child 1, pid 23461: no straight flush
Child 1, pid 23461: flush length 6 <D2,DA,DJ,DT,D5,D4>
Child 2, pid 23462: no straight flush
Child 3, pid 23463: no straight flush
Child 1, pid 23461: straight length 8 <SJ,CT,C9,S8,S7,C6,S5,H4>
Child 3, pid 23463: flush length 5 <SA,SQ,S9,S6,S3>
Child 2, pid 23462: flush length 6 <HA,HK,HT,H8,H6,H3>
Child 2, pid 23462: straight length 8 <H8,C7,H6,C5,S4,H3,S2,HA>
Child 3, pid 23463: straight length 6 <S6,H5,C4,S3,H2,SA>

```

Name your program **dealcard.c** and **submit it via BlackBoard on or before 11 February.**