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, you can create a program that will *generate **separate** child processes* to help dividing up a lot of processing tasks to be handled by each individual child. Effective division of labors among many processes actually forms the basis of ***cloud computing*** and ***big data processing***, making use of the computing power of multiple processors.

One common application to make use of abundant computing power is to perform *simulation study*. For simple questions concerning probabilistic results, mathematical analysis could be performed, e.g. the probability of getting 12 with two dice is 1/36 and the probability of getting 7 is 1/6. It is still possible to compute the probability of getting 40 with 10 dice. Likewise, one can compute the unconditional probability of playing for a 2/2 split in the bridge game with an analytic solution, but the conditional probability when some of the cards are known will deviate from the analytic unconditional probability. One way to estimate this probability is to *simulate*. In other words, generate many hands, say *N*, under the existing constraints and count the number of hands, *h*, satisfying the required condition. The required probability will then be *h*/*N*. Simple simulation could also be applied to study queuing systems, e.g. bank tellers serving customers, traffic congestion situation at Cross Harbor Tunnel. More complex computer simulation could be applied to study the infection model in epidemic study, typhoon trajectory prediction in meteorology, stock price change in financial technology, or pilot training in flight simulation.

In this exercise, we are to study the impact of waiting queue building up for a service provided by the teller of a bank. Customers arrive randomly and queue up for teller service. Tellers take on customers and serve them, one after the other. There is also variation in the service time by each teller. We are to study how the queue is built up, and as a result, the impact on the waiting time of the customers. To simplify the scenario, let us assume that there is *only one single queue*, and *only one single teller* for the customers. You are given input data about the customer arrival pattern and the servicing pattern by the teller in the *debug mode* of the program and would generate the patterns in the *simulation mode*.

In reality, random numbers are used to generate these patterns and are the key to simulation study. They are generated via a *pseudo-random number generator*, which is able to generate seemingly "random" numbers good enough for most practical applications. Yet, it is able to *repeat* the same sequence of numbers generated given the same ***seed***. That is very useful in simulation study for cross-comparison and for debugging. The most common generator is the *linear congruential generator*, provided in standard C library, **`rand()`**. You are to make use of a pseudo-random number generator by controlling the seed, thus the outcome. The outcomes are *deterministic* and this would facilitate debugging. If you want the program to really randomly generate patterns, set the seed to be the *current time* (which would differ each time the program is executed; see **`time_t`** above with the **`tv_usec`** field).

```
#include <stdlib.h>
  . . .
  int seed = 2023;
  . . .
  srand(seed); // do this only once and before using rand() to generate
  . . .
  while . . . {
      . . .
      r = rand() % N; // generate a random integer from 0 to N-1
      . . .
  }
  . . .
```

Data being used for the patterns can be generated randomly. For example, customer arrival events could follow a Poisson distribution, and the teller servicing events could follow a normal distribution. The standard queuing model usually assumes Poisson arrival and Poisson service in order to be analyzed mathematically, since Poisson events exhibit very nice mathematical properties. To achieve this, include the following code segment in your program, and make calls to the generator for the arrival and service patterns, after making appropriate modifications. Note that besides including the header file **`math.h`**, there is also a need to compile through linking to the **`maths`** library: **`cc -lm gen.c`**

```
#include <stdlib.h>
#include <math.h> // in order to use log() function
  . . .
  int seed = 2023;
  int arrivalPattern[100];
  int servicePattern[100];
  . . .
  srand(seed); // do this only once and before using rand() to generate
  . . .
  // generate patterns for 100 customers
  generate(arrivalPattern,100,5.0); // arriving every 5 time units
  arrivalPattern[0] = 0; // set default arrival time for first customer
  generate(servicePattern,100,4.0); // servicing average 4 time units
  // your computation
  . . .
```

```
int generate(int pattern[], int num, float timespan)
{
  int i;
  float u, r;

  for (i = 0; i < num; i++) {
     u = (float)rand() / RAND_MAX; // generate random number in (0,1]
     r = -log(u)*timespan; // exponential random variable for Poisson events
     pattern[i] = (int)r + 1; // round up to integer
  }
}
```

In this exercise, there are ***n*** child processes. The parent is just like a *server* that accepts user request and passes it to a child process to do the simulation. 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()`** is executed. To facilitate program development and debugging, the program would be executed in different *modes*, as indicated by the ***first*** argument. Depending on the mode, other arguments should be interpreted accordingly. The modes to be provided include *debug mode* (**D**) and *simulation mode* (**S**). You can assume that there are no errors in the user input argument list. Do not forget to ***wait*** for the child to complete to avoid *zombie processes*.

In the ***debug mode***, there will be additional arguments for the two patterns, separated by the sentinel value of **`-1`**. Note that the length of the two patterns should be the same if the input is correct. Only one child process needs to be created in this mode. The first customer is assumed to arrive at time 0.

In the ***simulation mode***, there will be a ***second*** argument to indicate the number of child processes, ***n***. This is followed by the ***t*** tasks. Each of the *t* tasks is specified by 4 arguments: the number of customers needing service, the seed to the random number generator to generate the arrival and service patterns, and then the arrival rate and service rate for generating the patterns. Each task is performed by each child in a *round-robin* manner. If there are more tasks than the number of child processes, some children would take up more than one task. The simulations are only performed by child processes, and the parent is not doing any real work. Each child will finally compute the average queue length and waiting time for generated customers within each task. In short, there will be 2+4*t* arguments in this mode, with *n* child processes being created. The first customer is also assumed to arrive at time 0.

In this exercise, the child processes are producing results independently. A better solution is to let each child report the result back. Then the parent process will *consolidate* together the results computed by the children, e.g. computing perhaps the mean and standard deviation of the waiting times and queue lengths. Remember that more attempts will lead to more accurate simulation. That is the concept behind *cloud computing*, based on the famous ***map/reduce*** paradigm. Inputs/simulation requests are divided up and *mapped* to different child processes; the parent then gets back the partial results and consolidates

them via *reduction*. That would require the ability that the child processes can report the computed information back to the parent (i.e., ability of inter-process communication). These complicated arrangements are ***not*** required in this exercise.

Hint: This is a highly simplified case for the *discrete event simulation* approach. All time units are in integers in this case. You could simply advance the clock for each time unit, and then check for the occurrence of the three key events: *customer arrival* (and queues up), *customer completion* (and departs), *customer welcomed* by the teller (and receives service). Once detected, appropriate actions can then be carried out for the specific event at that time unit.

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 concurrently*** 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. It is quite likely that those tasks with longer pattern files or more customers would execute more slowly.

Sample inputs and outputs:

```
queue D 0 1 4 2 -1 2 4 3 2
Parent, pid 12345: debug mode
Child 1, pid 12346: 4 customers arriving at 0 1 5 7
Child 1, pid 12346: 4 customers requiring service for 2 4 3 2
Child 1, pid 12346: time 0 customer 1 arrives, customer 1 waits for 0, queue length 0
Child 1, pid 12346: time 1 customer 2 arrives, queue length 1
Child 1, pid 12346: time 2 customer 1 departs, customer 2 waits for 1, queue length 0
# just for debugging, no need for output: Child 1, pid 12346: time 3 queue length 0
# just for debugging, no need for output: Child 1, pid 12346: time 4 queue length 0
Child 1, pid 12346: time 5 customer 3 arrives, queue length 1
Child 1, pid 12346: time 6 customer 2 departs, customer 3 waits for 1, queue length 0
Child 1, pid 12346: time 7 customer 4 arrives, queue length 1
# just for debugging, no need for output: Child 1, pid 12346: time 8 queue length 1
Child 1, pid 12346: time 9 customer 3 departs, customer 4 waits for 2, queue length 0
# just for debugging, no need for output: Child 1, pid 12346: time 10 queue length 0
Child 1, pid 12346: time 11 customer 4 departs, queue length 0
Child 1, pid 12346: all customers served at time 11
Child 1, pid 12346: maximum queue length 1
Child 1, pid 12346: average queue length 0.364
Child 1, pid 12346: total waiting time 4
Child 1, pid 12346: average waiting time 1.000
Child 1, pid 12346: child 1 completed execution
Parent, pid 12345: child 1 completed execution
```

```
queue D 0 1 4 2 3 1 2 3 -1 2 4 3 2 6 3 5 1
Parent, pid 12348: debug mode
Child 1, pid 12349: 8 customers arriving at 0 1 5 7 10 11 13 16
Child 1, pid 12349: 8 customers requiring service for 2 4 3 2 6 3 5 1
Child 1, pid 12349: time 0 customer 1 arrives, customer 1 waits for 0, queue length 0
Child 1, pid 12349: time 1 customer 2 arrives, queue length 1
Child 1, pid 12349: time 2 customer 1 departs, customer 2 waits for 1, queue length 0
Child 1, pid 12349: time 5 customer 3 arrives, queue length 1
Child 1, pid 12349: time 6 customer 2 departs, customer 3 waits for 1, queue length 0
Child 1, pid 12349: time 7 customer 4 arrives, queue length 1
Child 1, pid 12349: time 9 customer 3 departs, customer 4 waits for 2, queue length 0
Child 1, pid 12349: time 10 customer 5 arrives, queue length 1
Child 1, pid 12349: time 11 customer 4 departs, customer 6 arrives, customer 5 waits for 1, queue
length 1
Child 1, pid 12349: time 13 customer 7 arrives, queue length 2
Child 1, pid 12349: time 16 customer 8 arrives, queue length 3
Child 1, pid 12349: time 17 customer 5 departs, customer 6 waits for 6, queue length 2
Child 1, pid 12349: time 20 customer 6 departs, customer 7 waits for 7, queue length 1
Child 1, pid 12349: time 25 customer 7 departs, customer 8 waits for 9, queue length 0
Child 1, pid 12349: time 26 customer 8 departs, queue length 0
Child 1, pid 12349: all customers served at time 26
Child 1, pid 12349: maximum queue length 3
Child 1, pid 12349: average queue length 1.038
Child 1, pid 12349: total waiting time 27
Child 1, pid 12349: average waiting time 3.375
Child 1, pid 12349: child 1 completed execution
Parent, pid 12348: child 1 completed execution
```

```
queue D 0 1 2 3 2 4 3 2 -1 1 4 1 1 2 4 2 1
Parent, pid 12355: debug mode
Child 1, pid 12356: 8 customers arriving at 0 1 3 6 8 12 15 17
Child 1, pid 12356: 8 customers requiring service for 1 4 1 1 2 4 2 1
Child 1, pid 12356: time 0 customer 1 arrives, customer 1 waits for 0, queue length 0
Child 1, pid 12356: time 1 customer 1 departs, customer 2 arrives, customer 2 waits for 0, queue
length 0
Child 1, pid 12356: time 3 customer 3 arrives, queue length 1
Child 1, pid 12356: time 5 customer 2 departs, customer 3 waits for 2, queue length 0
Child 1, pid 12356: time 6 customer 3 departs, customer 4 arrives, customer 4 waits for 0, queue
length 0
Child 1, pid 12356: time 7 customer 4 departs, queue length 0
Child 1, pid 12356: time 8 customer 5 arrives, customer 5 waits for 0, queue length 0
Child 1, pid 12356: time 10 customer 5 departs, queue length 0
Child 1, pid 12356: time 12 customer 6 arrives, customer 6 waits for 0, queue length 0
Child 1, pid 12356: time 15 customer 7 arrives, queue length 1
Child 1, pid 12356: time 16 customer 6 departs, customer 7 waits for 1, queue length 0
Child 1, pid 12356: time 17 customer 8 arrives, queue length 1
Child 1, pid 12356: time 18 customer 7 departs, customer 8 waits for 1, queue length 0
Child 1, pid 12356: time 19 customer 8 departs, queue length 0
Child 1, pid 12356: all customers served at time 19
Child 1, pid 12356: maximum queue length 1
Child 1, pid 12356: average queue length 0.211
Child 1, pid 12356: total waiting time 4
Child 1, pid 12356: average waiting time 0.500
Child 1, pid 12356: child 1 completed execution
Parent, pid 12355: child 1 completed execution
```

```
queue S 3 100 2023 5.0 4.0 200 2024 4.0 5.0
Parent, pid 23456: 3 children, 2 tasks, simulation mode
Child 1, pid 23457: simulating 100 customers, seed 2023, arrival 5.0, service 4.0
Child 2, pid 23458: simulating 200 customers, seed 2024, arrival 4.0, service 5.0
Child 1, pid 23457: all customers served at time 593
Child 1, pid 23457: maximum queue length 7
Child 1, pid 23457: average queue length 1.094
Child 1, pid 23457: total waiting time 649
Child 1, pid 23457: average waiting time 6.490
Child 1, pid 23457: child 1 completed execution
Parent, pid 23456: child 1 completed execution
Child 2, pid 23458: all customers served at time 1040
Child 2, pid 23458: maximum queue length 34
Child 2, pid 23458: average queue length 17.441
Child 2, pid 23458: total waiting time 18139
Child 2, pid 23458: average waiting time 90.695
Child 2, pid 23458: child 2 completed execution
Parent, pid 23456: child 2 completed execution
```

```
queue S 3 1000 2023 5.0 5.0 1500 2024 5.0 4.8 2000 2025 5.0 4.8 1000 2026 10.0 8.0 1500 2027 10.0
8.0 2000 2028 4.0 2.0 1000 2029 4.0 3.0
Parent, pid 23456: 3 children, 7 tasks, simulation mode
Child 1, pid 24322: simulating 1000 customers, seed 2023, arrival 5.0, service 5.0
Child 2, pid 24323: simulating 1500 customers, seed 2024, arrival 5.0, service 4.8
Child 3, pid 24324: simulating 2000 customers, seed 2025, arrival 5.0, service 4.8
Child 1, pid 24322: all customers served at time 5778
Child 1, pid 24322: maximum queue length 44
Child 1, pid 24322: average queue length 21.296
Child 1, pid 24322: total waiting time 123051
Child 1, pid 24322: average waiting time 123.051
Child 1, pid 24322: simulating 1000 customers, seed 2026, arrival 10.0, service 8.0
Child 2, pid 24323: all customers served at time 8254
Child 2, pid 24323: maximum queue length 73
Child 2, pid 24323: average queue length 34.114
Child 2, pid 24323: total waiting time 281578
Child 2, pid 24323: average waiting time 187.719
Child 2, pid 24323: simulating 1500 customers, seed 2027, arrival 10.0, service 8.0
Child 3, pid 24324: all customers served at time 11136
Child 3, pid 24324: maximum queue length 78
Child 3, pid 24324: average queue length 24.656
Child 3, pid 24324: total waiting time 274568
Child 3, pid 24324: average waiting time 137.284
Child 3, pid 24324: simulating 2000 customers, seed 2028, arrival 4.0, service 2.0
Child 1, pid 24322: all customers served at time 10315
Child 1, pid 24322: maximum queue length 23
Child 1, pid 24322: average queue length 4.018
Child 1, pid 24322: total waiting time 41441
```

```
Child 1, pid 24322: average waiting time 41.441
Child 1, pid 24322: simulating 1000 customers, seed 2029, arrival 4.0, service 3.0
Child 3, pid 24324: all customers served at time 8934
Child 3, pid 24324: maximum queue length 7
Child 3, pid 24324: average queue length 0.414
Child 3, pid 24324: total waiting time 3698
Child 3, pid 24324: average waiting time 1.849
Child 3, pid 24324: child 3 completed execution
Parent, pid 24321: child 3 completed execution
Child 1, pid 24322: all customers served at time 4478
Child 1, pid 24322: maximum queue length 10
Child 1, pid 24322: average queue length 1.587
Child 1, pid 24322: total waiting time 7108
Child 1, pid 24322: average waiting time 7.108
Child 1, pid 24322: child 1 completed execution
Parent, pid 24321: child 1 completed execution
Child 2, pid 24323: all customers served at time 15568
Child 2, pid 24323: maximum queue length 13
Child 2, pid 24323: average queue length 2.312
Child 2, pid 24323: total waiting time 35992
Child 2, pid 24323: average waiting time 23.995
Child 2, pid 24323: child 2 completed execution
Parent, pid 24321: child 2 completed execution
```

Level 1 requirement: the child processes are created properly and can display the correct information about the given arrival and service patterns.
Level 2 requirement: the unique child process can produce partially correct results in the debug mode.
Level 3 requirement: the unique child process can product almost all correct results in the debug mode.
Level 4 requirement: all child processes can work correctly under all the cases and produce correct simulation results in terms of average queue length and waiting time.
Bonus level: you can handle the simulation with more than one teller. There can be several possible queuing strategies for the multiple tellers. Please explain briefly your adopted strategy.

Name your program **queue.c** and **submit it via BlackBoard on or before 3 March 2023**.