CS 343 Winter 2024 – Assignment 6
Instructor: Colby Parsons
Due Date: Monday, April 8, 2024 at *23:55 NO LATE*

March 19, 2024

This assignment continues task communication in μC++ and high-level techniques for structuring complex interactions among tasks. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution.

WATCola is renown for its famous line of healthy soda pop, which come in the dazzling array of flavours: Blues Black-Cherry, Classic Cream-Soda, Rock Root-Beer, and Jazz Lime. The company recently won the contract to provide soda to vending machines on campus. The vending machines are periodically restocked by a WATCola truck making deliveries from the local bottling plant.

This assignment simulates a simple concession service using the objects and relationships in Figure 1. (Not all possible communication paths are shown in the diagram.)

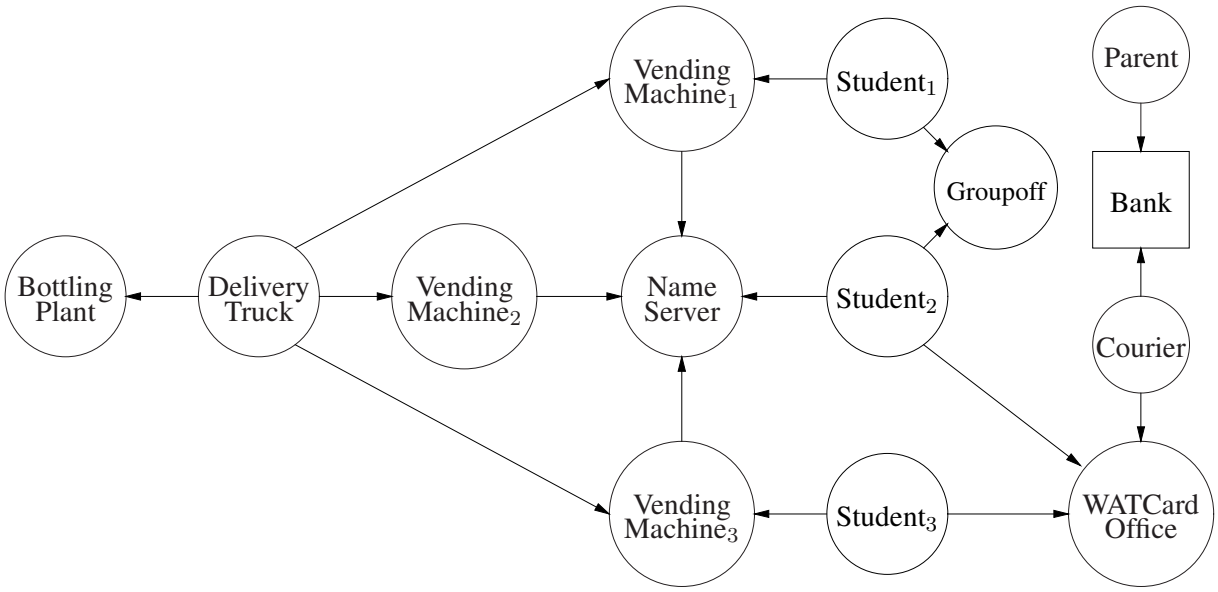

Figure 1: Concession Service

The following constants are used to configure the simulation, and are read from a text file:

```
SodaCost              2   # Manufacturer Suggested Retail Price (MSRP) per bottle
NumStudents           2   # number of students to create
MaxPurchases          8   # maximum number of bottles a student purchases
NumVendingMachines    3   # number of vending machines
MaxStockPerFlavour    5   # maximum number of bottles of each flavour in a vending machine
MaxShippedPerFlavour  3   # maximum number of bottles of each flavour generated by the bottling
                          #     plant per production run
TimeBetweenShipments  3   # length of time between shipment pickup
GroupoffDelay         10  # length of time between initializing gift cards
ParentalDelay         5   # length of time between deposits of funds
NumCouriers           1   # maximum number of couriers in the pool
```

Comments in the file (from # to the end-of-line), as well as blank lines, are ignored. The constants may appear in any order. Any number of spaces/tabs may appear around a constant name, value or comment. Assume each constant appears in the configuration file, is syntactically correct, its value is within an appropriate range (i.e., no error checking is required), and only one constant is defined per line. Modify this sample file to obtain interesting results when testing.

The following types and routines are required in the assignment (you may add only a public destructor and private/protected members):

1. ```
   struct ConfigParms {
       unsigned int sodaCost;                  // MSRP per bottle
       unsigned int numStudents;               // number of students to create
       unsigned int maxPurchases;              // maximum number of bottles a student purchases
       unsigned int numVendingMachines;        // number of vending machines
       unsigned int maxStockPerFlavour;        // maximum number of bottles of each flavour stocked
       unsigned int maxShippedPerFlavour;      // number of bottles of each flavour in a shipment
       unsigned int timeBetweenShipments;      // length of time between shipment pickup
       unsigned int groupoffDelay;             // length of time between initializing gift cards
       unsigned int parentalDelay;             // length of time between cash deposits
       unsigned int numCouriers;               // number of couriers in the pool
   };

   void processConfigFile( const char * configFile, ConfigParms & cparms );
   ```

   Routine processConfigFile is called by the main program to read and parse the configuration file, and places the parsed values into the fields of the argument for output parameter cparms.

2. ```
   _Task Student {
       void main();
     public:
       Student( Printer & prt, NameServer & nameServer, WATCardOffice & cardOffice, Groupoff & groupoff,
                unsigned int id, unsigned int maxPurchases );
   };
   ```

   A Student's function is to periodically buy some of their favourite soda from a vending machine (a bottle costs whatever the vending machine is charging). Each student is passed an id in the range [0, NumStudents) for identification. A student begins by selecting a random number of bottles to purchase [1, MaxPurchases], a random favourite flavour [0, 3] (see 4 flavours on page 1), creates a WATCard from the WATCardOffice with a $5 balance, creates a gift card from Groupoff with a value of SodaCost, and obtains the location of a vending machine from the name server. A student terminates after purchasing their selected random number of bottles of their favourite flavour.

   Before each attempt to purchase a soda (including the first), a student yields a random number of times in the range [1, 10]. A student then wants to buy a soda but may need to wait for money to be transferred either from the WATCardOffice to their WATCard or from Groupoff to their gift card. Once money is available on either card, the student can attempt to buy a bottle of soda from their current vending machine; if both cards have money, use the gift card first. **Note, a giftcard future can only be used once, so it must be *reset* after use to prevent any further usage.** If a courier has lost a student's WATCard during a transfer (see WATCardOffice::Courier), the exception WATCardOffice::Lost is raised when the future value is accessed. In this case, the student must create a new WATCard from the WATCardOffice with a $5 balance, and re-attempt to purchase a soda **without yielding as a purchase has not occurred**. Note, a courier can lose a student's WATCard during during each transfer for the new WATCard so this issue can occur repeatedly.

   Once money is available, a purchase begins with a student calling a vending machine to begin a transaction. If the vending machine debits the WATCard and delivers a bottle of soda, the student drinks it and attempts another purchase. If the vending machine delivers a free bottle of soda (which does not count as a purchased soda), there is a 50% chance the student watches an advertisement associated with it by yielding 4 times (not random) while drinking the free soda and then attempts another purchase. If the vending machine indicates insufficient funds, a student transfers the current vending-machine soda-cost plus $5 to their WATCard via the WATCard office and attempts another purchase. If the vending machine is out of the student's favourite flavour, the student must obtain a new vending machine from the name server and attempt another purchase. Hence, a student may perform busy-wait among vending machines until their specific soda appears from the bottling plant. ***Watch out for the case of a student who only buys one soda using the gift card.***

3.
```
class WATCard {
    WATCard( const WATCard & ) = delete;     // prevent copying
    WATCard( const WATCard && ) = delete;
    WATCard & operator=( const WATCard & ) = delete;
    WATCard & operator=( const WATCard && ) = delete;
  public:
    typedef Future_ISM<WATCard *> FWATCard; // future watcard pointer
    WATCard();
    void deposit( unsigned int amount );
    void withdraw( unsigned int amount );
    unsigned int getBalance();
};
```

The WATCard manages the money associated with a card. When a WATCard is created, it has a $0 balance. A courier calls deposit after a funds transfer. A vending machine calls withdraw when a soda is purchased. A student and a vending machine call getBalance to determine the balance. FWATCard is a future pointer to a student's WATCard for synchronizing access to the WATCard between the student and the courier.

4.
```
_Task WATCardOffice {
    struct Job {                                // marshalled arguments and return future
        Args args;                              // call arguments (YOU DEFINE "Args")
        WATCard::FWATCard result;               // return future
        Job( Args args ) : args( args ) {}
    };
    _Task Courier { ... };                      // communicates with bank

    void main();
  public:
    _Exception Lost {};                             // lost WATCard
    WATCardOffice( Printer & prt, Bank & bank, unsigned int numCouriers );
    WATCard::FWATCard create( unsigned int sid, unsigned int amount )
        __attribute__(( warn_unused_result ));
    WATCard::FWATCard transfer( unsigned int sid, unsigned int amount, WATCard * card )
        __attribute__(( warn_unused_result ));
    Job * requestWork() __attribute__(( warn_unused_result ));
};
```

The WATCardOffice is an administrator task used by a student to transfer funds from their bank account to their WATCard to buy a soda. Initially, the WATCard office creates a fixed-sized courier pool with numCouriers courier tasks to communicate with the bank. (Additional couriers may *not* be created after the WATCardOffice begins.) A student performs an asynchronous call to create to create a "real" WATCard with an initial balance. A future WATCard is returned and sufficient funds are subsequently obtained from the bank (see Parent task) via a courier to satisfy the create request. A student performs an asynchronous call to transfer when their WATCard indicates there is insufficient funds to buy a soda. Again, a future WATCard is returned and sufficient funds are subsequently obtained from the bank as for create. The WATCard office is empowered to transfer funds from a student's bank-account to its WATCard by sending a request through a courier to the bank. Each courier task calls requestWork, blocks until a Job request is ready, and then receives the next Job request as the result of the call. As soon as the request is satisfied (i.e., money is obtained from the bank), the courier updates the student's WATCard. There is a 1 in 6 chance, prng(6) == 0, a courier loses a student's WATCard after the update. When the card is lost, the exception WATCardOffice::Lost is inserted into the future, rather than making the future available, and the current WATCard is deleted.

5.
```
_Monitor Bank {
  public:
    Bank( unsigned int numStudents );
    void deposit( unsigned int id, unsigned int amount );
    void withdraw( unsigned int id, unsigned int amount );
};
```

The Bank is a monitor, which behaves like a server, that manages student-account information for all students. Each student's account initially starts with a balance of $0. The parent calls deposit to endow gifts to a specific

student. A courier calls withdraw to transfer money on behalf of the WATCard office for a specific student. The courier waits until enough money has been deposited, which may require multiple deposits.

6.
```
_Task Parent {
    void main();
  public:
    Parent( Printer & prt, Bank & bank, unsigned int numStudents, unsigned int parentalDelay );
};
```

The Parent task periodically gives a random amount of money [$1, $3] to a random student [0, NumStudents), in that order. Before each gift is transferred (including the first), the parent yields for parentalDelay times (not random). The parent must check for a call to its destructor to know when to terminate. Since it must not block on this call, it is necessary to use a terminating **_Else** on the accept statement. Hence, the parent is performing a yielding busy-wait on calls to its destructor.

7.
```
_Task Groupoff {
    void main();
  public:
    Groupoff( Printer & prt, unsigned int numStudents, unsigned int sodaCost, unsigned int groupoffDelay );
    WATCard::FWATCard giftCard();
};
```

The Groupoff task begins by first accepting a call **from all students** to obtain a future gift-card. Afterwhich groupoff periodically creates and puts a real WATCard with value SodaCost into a random future gift-card. A future gift-card is assigned only once. Before each future gift-card is assigned a real WATCard, groupoff yields for groupoffDelay times (not random). The groupoff loops until all the future gift-cards are assigned a real WATCard or a call to its destructor occurs. Since it must not block on the destructor call, it is necessary to use a terminating **_Else** on the accept statement. Note, this use of **_Else** is not busy waiting because there are a finite number of students.

8.
```
_Task VendingMachine {
    void main();
  public:
    _Exception Free {};                          // free, advertisement
    _Exception Funds {};                         // insufficient funds
    _Exception Stock {};                         // flavour out of stock
    VendingMachine( Printer & prt, NameServer & nameServer, unsigned int id, unsigned int sodaCost );
    void buy( BottlingPlant::Flavours flavour, WATCard & card );
    unsigned int * inventory() __attribute__(( warn_unused_result ));
    void restocked();
    _Nomutex unsigned int cost() const;
    _Nomutex unsigned int getId() const;
};
```

A vending machine's function is to sell soda to students at some cost. Each vending machine is passed an id in the range [0, NumVendingMachines) for identification, and the MSRP price for a bottle of soda. A new vending machine is empty (no stock) and begins by registering with the name server.

A student calls buy to obtain one of their favourite sodas. The vending machine checks first if the student has sufficient funds to purchase the soda and second if the specified soda is available; then the student's WATCard is debited by the cost of a soda. Otherwise, raise exceptions Funds or Stock, respectively. Once a purchase is possible, there is a 1 in 5 chance the soda is free, if the student watches an advertisement, which is indicated by raising exception Free. Use concurrent exceptions to raise Funds, Stock or Free on the correct task.

Periodically, the truck comes by to restock the vending machines with new soda from the bottling plant. Restocking is performed in two steps. The truck calls inventory to return a pointer to an array containing the amount of each kind of soda currently in the vending machine. The truck uses this information to transfer as much of its stock as fits into the machine; for each kind of soda, no more than MaxStockPerFlavour per flavour can be added to a machine. If the truck cannot top-up a particular flavour because there is insufficient stock, it transfers as many bottles as it has (which could be 0). After transferring new soda into the machine by directly modifying the array passed from inventory, the truck calls restocked to indicate the operation is complete. The vending

machine cannot accept buy calls during restocking. The cost member returns the cost of purchasing a soda for this machine. The getId member returns the identification number of the vending machine.

9.
```
_Task NameServer {
    void main();
  public:
    NameServer( Printer & prt, unsigned int numVendingMachines, unsigned int numStudents );
    void VMregister( VendingMachine * vendingmachine );
    VendingMachine * getMachine( unsigned int id ) __attribute__(( warn_unused_result ));
    VendingMachine ** getMachineList() __attribute__(( warn_unused_result ));
};
```

The NameServer is a server task used to manage the vending-machine names. The name server is passed the number of vending machines, NumVendingMachines, and the number of students, NumStudents. It begins by logically distributing the students evenly across the vending machines in a round-robin fashion. That is, student id 0 is assigned to the first registered vending-machine, student id 1 is assigned to the second registered vending-machine, etc., until there are no more registered vending-machines, and then start again with the first registered vending-machine (i.e., round robin). Vending machines call VMregister to register themselves so students can subsequently locate them. A student calls getMachine to find a vending machine, and the name server must cycle through the vending machines *separately* for each student starting from the initial position via modulo incrementing to ensure a student has a chance to visit every machine. The truck calls getMachineList to obtain an array of pointers to vending machines so it can visit each machine to deliver new soda. All vending machines are registered **before** accepting calls to other members.

10.
```
_Task BottlingPlant {
    void main();
  public:
    enum Flavours { ..., NUM_OF_FLAVOURS }; // flavours of soda (YOU DEFINE)
    _Exception Shutdown {};                  // shutdown plant
    BottlingPlant( Printer & prt, NameServer & nameServer, unsigned int numVendingMachines,
                   unsigned int maxShippedPerFlavour, unsigned int maxStockPerFlavour,
                   unsigned int timeBetweenShipments );
    void getShipment( unsigned int cargo[] );
};
```

The bottling plant periodically produces random new quantities of each flavour of soda, [0, MaxShipped PerFlavour] per flavour. The bottling plant is passed the number of vending machines, NumVendingMachines, the maximum number of bottles of each flavour generated during a production run and subsequently shipped, MaxShippedPerFlavour, the maximum number of bottles of each flavour in a vending machine MaxStockPerFlavour, and the length of time between shipment pickups by the truck, TimeBetweenShipments. It begins by creating a truck, performing a production run, and waiting for the truck to pickup the production run. The truck then distributes these bottles to initialize the registered vending machines. To simulate a production run of soda, the bottling plant yields for TimeBetweenShipments times (not random). The truck calls getShipment to obtain a shipment from the plant (i.e., the production run), and the shipment is copied into the cargo array passed by the truck. getShipment propagates the exception Shutdown if the bottling plant is closing down, and no shipment is copied into the cargo array passed by the truck. Use concurrent exceptions to raise Shutdown on the correct task. The bottling plant does not start another production run until the truck has picked up the current run. Define the public type Flavours to represent the different soda flavours available on page 1.

11.
```
_Task Truck {
    void main();
  public:
    Truck( Printer & prt, NameServer & nameServer, BottlingPlant & plant,
           unsigned int numVendingMachines, unsigned int maxStockPerFlavour );
};
```

The truck moves soda from the bottling plant to the vending machines. The truck is passed the number of vending machines, numVendingMachines, and the maximum number of bottles of each flavour in a vending machine maxStockPerFlavour. The truck begins by obtaining the location of each vending machine from the

name server. Before each shipment from the bottling plant, the truck yields a random number of times [1, 10] to get a coffee from Tom Hortons. The truck then calls BottlingPlant::getShipment to obtain a new shipment of soda; any soda still on the truck is thrown away as it is past its due date. If the bottling plant is closing down, the truck terminates. To ensure fairness, the vending machines are restocked in cyclic order starting at the vending machine *after* the last machine the truck restocked, until there is no more soda on the truck or the truck has made a complete cycle of all the vending machines; so there is no guarantee each vending machine is completely restocked or the entire complement of vending machines is restocked or all the soda on the truck is used. The truck can only restock up to MaxStockPerFlavour for each flavour in each vending machine (see VendingMachine task). Finally, after a delivery to each vending machine, there is a 1 in 100 chance of a flat tire, which takes 10 yields (not random) to fix.

12. **_Monitor** / **_Cormonitor** Printer {

```
public:
    enum Kind { Parent, Groupoff, WATCardOffice, NameServer, Truck, BottlingPlant, Student, Vending, Courier };
    Printer( unsigned int numStudents, unsigned int numVendingMachines, unsigned int numCouriers );
    void print( Kind kind, char state );
    void print( Kind kind, char state, unsigned int value1 );
    void print( Kind kind, char state, unsigned int value1, unsigned int value2 );
    void print( Kind kind, unsigned int lid, char state );
    void print( Kind kind, unsigned int lid, char state, unsigned int value1 );
    void print( Kind kind, unsigned int lid, char state, unsigned int value1, unsigned int value2 );
};
```

All output from the program is generated by calls to a printer, excluding error messages. The printer generates output like that in Figure 2. Each column is assigned to a particular kind of object. There are 9 kinds of objects: parent, groupoff, WATCard office, name server, truck, bottling plant, student, vending machine, and courier. Student, vending machine, and courier have multiple instances. For the objects with multiple instances, these objects pass in their local identifier [0,N) when printing. Each kind of object prints specific information in its column:

- The parent prints the following information:

State	Meaning	Additional Information
S	starting	
D s,g	deposit gift	student s receiving gift, amount of gift g
F	finished	

- The groupoff prints the following information:

State	Meaning	Additional Information
S	starting	
D g	deposit gift	amount of gift g
F	finished	

- The WATCard office prints the following information:

State	Meaning	Additional Information
S	starting	
W	request work call complete	
C s,a	create call complete	student s, transfer amount a
T s,a	transfer call complete	student s, transfer amount a
F	finished	

- The name server prints the following information:

State	Meaning	Additional Information
S	starting	
R v	register vending machine	vending machine v registering
N s,v	new vending machine	student s requesting vending machine, new vending machine v
F	finished	

```
$ soda soda.config 55718 1
Parent   Gropoff  WATOff  Names   Truck   Plant   Stud0   Stud1   Mach0   Mach1   Mach2   Cour0
*******  *******  *******  *******  *******  *******  *******  *******  *******  *******  *******  *******
S        S        S       S                                                                        S
                          R0                                       S2
D0,3                      R1                                               S2
                          R2               S       S3,3                            S2
                  C0,5    N0,0     S       G6      V0      S3,2                            t0,5
                  W                P6
                                   d0,6
                                   U0,14
D1,2              C1,5    N1,1     D0,0    P               V1      r
D1,2                               P9      G9                      R
                                   d1,9
                                   U1,11
D1,3     D2                        D1,0    P                               r
                                           G8                              R
D1,1                                                       a3,2            A
D1,1                               P8                      a3,2            A
                                   d2,8
                                   U2,12
         D2                        D2,0
         F                                  P                                       r
D1,1                               P4      G4      G3,0    G3,0    B3,0    B3,0    R
                                   d0,4
                                   U0,11
D1,2                               D0,0    P                       r
D1,1                               P6      G6                      R
                                   d1,6
                                   U1,8
D1,2                               D1,0    P                               r
D0,3                                       G11                             R               T0,5
                                                                                           t1,5
                                   P11                                                     T1,5
                                   d2,11
                                   U2,3
                  W                D2,2    P       V1      B3,3                    r
D1,2                      N0,1     d0,2    G5              F               B3,1    R
                                   U0,9
                                   D0,0            B3,3            r
D1,3                      N0,2     P5              V2              R       B3,0
                                   d1,5
                                   U1,6
D0,3                               D1,1    P                               r
                                   d2,1
                                   U2,3
                                   D2,1    G9                              R       r
                                   d0,1
                                   U0,8
D0,3                               D0,0                            r               R
                                                   B3,1            R
D0,1                               F       F       F               F       F       B3,3
D1,3                      F                                                        F
D1,3              F                                                                        F
F
**********************
```

Figure 2: WATCola : Example Output

- The truck prints the following information:

State	Meaning	Additional Information
S	starting	
P a	picked up shipment	total amount a of all sodas in the shipment
d v,r	begin delivery to vending machine	vending machine v, total amount remaining r in the shipment
U v,b	unsuccessfully filled vending machine	vending machine v, total number of bottles b not replenished
D v,r	end delivery to vending machine	vending machine v, total amount remaining r in the shipment
W	wait for repair of flat tire	
F	finished	

States d and D are printed for each vending machine visited during restocking.

- The bottling plant prints the following information:

State	Meaning	Additional Information
S	starting	
G b	generating soda	bottles b generated in production run
P	shipment picked up by truck	
F	finished	

- A student prints the following information:

State	Meaning	Additional Information
S f,b	starting	favourite soda f, number of bottles b to purchase
V v	selecting vending machine	vending machine v selected
G f,b	gift-card soda	soda flavour f purchased, giftcard balance b
B f,b	bought soda	soda flavour f purchased, WATCard balance b
a f,b	free soda, advertisement	soda flavour f purchased, giftcard balance b
A f,b	free soda, advertisement	soda flavour f purchased, WATCard balance b
X	did not watch advertisement	
L	WATCard lost	
F	finished	

- A vending machine prints the following information:

State	Meaning	Additional Information
S c	starting	cost c per bottle
r	start reloading by truck	
R	complete reloading by truck	
A	free soda, advertisement	
B f,r	student bought a soda	soda flavour f purchased, amount remaining r of this flavour
F	finished	

- A courier prints the following information:

State	Meaning	Additional Information
S	starting	
t s,a	start funds transfer	student s requesting transfer, amount a of transfer
L s	lost WATCard card	student s requesting transfer
T s,a	complete funds transfer	student s requesting transfer, amount a of transfer
F	finished	

Information is buffered until a column is overwritten for a particular entry, which causes the buffered data to be flushed. If there is no new stored information for a column since the last buffer flush, an empty column is printed. After an object has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in internal representation; **do not build and store strings of text for output**.

To obtain semi-repeatable results, all random numbers are generated using the μC++ task-member prng (see Appendix C in the μC++ reference manual).

The program main starts by calling processConfigFile to read and parse the simulation configurations. After which, it creates any additional processors for parallelism (compile with −multi) as follows:

```
uProcessor p[processors - 1];  // number of kernel threads
```

Since a program starts with one kernel thread, only processors − 1 additional kernel threads are necessary. It then creates in order: printer, bank, parent, WATCard office, groupoff, name server, vending machines, bottling plant, and students. The truck is created by the bottling plant; the couriers are created by the WATCard office. The program terminates once all of the students have purchased their specified number of bottles. Note, there is one trick in closing down the system: delete the bottling plant *before* deleting the vending machines to allow the truck to complete its final deliveries to the vending machines; otherwise, a deadlock can occur.

The executable program is named soda and has the following shell interface:

```
soda [ config-file | 'd' [ seed (> 0) | 'd' [ processors (> 0) | 'd' ] ] ]
```

config−file is the text (formatted) file containing the configuration constants. If d or no value for config−file is specified, assume file name soda.config.

seed is the starting seed for the random-number generator (> 0). If seed is specified, call set_seed(seed). If d or no value for seed is specified, do nothing as PRNG sets the seed to an arbitrary value.

processors is the number of processors for parallelism (> 0). If d or no value for processors is specified, assume 1.

Check all command arguments for correct form (integers) and range; print an appropriate usage message and terminate the program if a value is missing or invalid.

Do not try to code this program all at once. Write each task separately and test it before putting the pieces together. Play with the configuration file and sample executable to familiarize yourself with the system before starting to write code.

Submission Guidelines

This assignment should be done by a team of two people because of its size. Both people receive the same grade (no exceptions). **Only one member of a team submits the assignment.** The instructor and/or instructional-coordinator does not arbitrate team disputes; team members must handle any and all problems.

Follow these guidelines carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text or test-document file, e.g., *.{txt,testdoc} file, must be ASCII text and not exceed 750 lines in length, using the command fold −w120 *.testdoc | wc −l.** Programs must be divided into separate compilation units, i.e., *.{h,cc,C,cpp} files, where applicable. Use the submit command to electronically copy the following files to the course account.

1. config.{h,cc} – process configuration file (provided)
2. *.{h,cc,C,cpp} – code for the assignment. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
3. soda.testdoc – test documentation for 6–8 interesting assignment events, e.g., a courier losting a WATcard and how the student deals with it, which includes the input that generated the events. Extract from the output the relevant information that illustrates the interesting event and explain how this output demonstrates correct handling of the event.
4. group.txt – give the userid of each group member, with one userid per line (i.e., 2 lines in this file), e.g.:

   ```
   sjholmes
   jjwatson
   ```

   Do not submit this file if you are working by yourself.
5. Makefile – construct a makefile **similar to those presented in the course** to compile the program. **This makefile must NOT contain hand-coded dependencies.** The makefile is invoked as follows:

   ```
   $ make soda
   $ soda ...
   ```

   Put this Makefile in the directory with the program, name the source files as specified above, and enter the appropriate make to compile a specific version of the programs. This Makefile must be submitted with the assignment to build the program, so it must be correct. Use the web tool Request Test Compilation to ensure you have submitted the appropriate files, your makefile is correct, and your code compiles in the testing environment.

Follow these guidelines. Your grade depends on it!