Concurrent Programming Problems

The Cookie Jar Problem. A cookie jar that never becomes empty (the best type!) is being shared by two sisters, Tina and Judy, using the following (not-so-good) rule: Judy can get a cookie from the jar only after Tina (being the older sister) gets a cookie in at least two separate occasions, whereas Tina gets a cookie from the jar whenever she wants to. Treating Tina and Judy as two independently executing processes and the Cookie Jar as a shared resource, develop a concurrent algorithm to solve the Cookie Jar Problem.

The Finite-Source Cookie Jar Problem. A cookie jar is being shared by two sisters, Tina and Judy, using the following rule: Judy can get a cookie from the jar only after Tina (being the older sister) gets a cookie in at least two separate occasions, whereas Tina gets a cookie from the jar whenever she wants to. The mother is alerted by Judy or Tina whenever they want to eat a cookie and the jar is empty, in which case the mother fills the jar up (from a hidden source), and the kids go on eating. Treating Tina, Judy and the mother as three independently executing processes and the Cookie Jar as a shared resource, develop a concurrent algorithm to solve the Finite-Source Cookie Jar Problem.

The Savings Account Problem. A savings account is shared by several people (i.e., processes). Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative, and withdrawals are serviced First-Come-First-Serve with one twist: Those who want to withdraw money from the account are desperate to get their money; so, if the balance is not sufficient then they wait (in a FIFO order) until somebody else deposits enough money for them to make their withdrawals. Note that those waiting for others to deposit (so that they can withdraw) are serviced strictly FIFO. For example, suppose the current balance is 200 dollars, and customer A is waiting to withdraw 300 dollars. Assume another customer, say B, arrives with the request of withdrawing 200 dollars. Customer B must wait until customer A is serviced. Develop a concurrent algorithm to solve the Savings Account Problem.

The Roller Coaster Problem. Suppose there are n passenger processes and one (roller coaster) car process. The passengers repeatedly wait to take rides in the car, which can hold C passengers, $C < n$. However, the car can go around the tracks only when it is full. Develop codes for the actions of the car and passenger processes, and develop a solution to synchronize them.
Generalize your answer to the above part to employ m car processes, $m > 1$. Since there is only one track, cars cannot pass each other; i.e., they must finish going around the track in the order in which they started. Again, a car can only go around the tracks when it is full. Develop a concurrent algorithm to solve the Roller Coaster problem.

Fair Readers-Writers Problem. Give a semaphore-based solution to the **Fair Readers-Writers Problem where NOBODY STARVES:** When there are readers and writers waiting to read and write (due to a writer or reader in its CS), the service of the system becomes first-come-first-serve (FIFO) with the provision that consecutive FIFO readers are still allowed to read concurrently. You may assume that you know the total number of readers and writers in the system. Develop a concurrent algorithm to solve the Fair Readers-Writers Problem.

One-Lane Bridge problem. Cars coming from the north and south must pass a bridge across a river. Unfortunately, there is only one lane on the bridge. So, at any moment, the bridge can be crossed only by *one or more* cars coming from the same direction (but not from opposite directions). For liveness, the direction of traffic across the bridge changes each time 10 cars have crossed from one direction while one or more cars are waiting to cross it from the opposite direction. Give a concurrent algorithm to solve the One-Lane Bridge Problem.

Searchers/Inserters/Deleters Problem. Three kinds of processes share access to a singly-linked list: searchers, inserters, and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions. You are to write an algorithm to synchronize searcher, inserter and deleter processes.

Drinking Philosophers Problem. Consider the following generalization of the dining philosophers problem. Given is an undirected graph G. Philosophers are associated with nodes of the graph and can communicate only with neighbors. A bottle is associated with each edge of G. Each philosopher cycles between three states: tranquil, thirsty, and drinking. A tranquil philosopher may become thirsty. Before drinking, the philosopher must acquire the bottle associated with every edge connected to the philosopher's node. After drinking, a philosopher again becomes tranquil. Give a concurrent algorithm to this problem that is deadlock-free.

The Ski Lift Problem. The Holiday Valley Ski Resort in Ellicottville, NY (http://www.holidayvalley.com), has a quad chair, called Yodeler, that has four distinct loading lines: left-singles (LS) line, left-triples (LT) line, right singles (RS) line, and right triples (RT) line. (We know in reality that people do not line up in triples; instead, they line up sometimes in doubles, sometimes in triples, and sometimes in quadruples. This, however, adds to the complexity of our problem, and we will not deal with it).
Procedure to select the loading line: A Yodeler skier (i.e., concurrent process), upon arriving at the bottom of the lift for a ride, looks at the length (i.e., the number of people) of each line, and
(a) (s)he chooses the LS line if the length of the LS line is less than (i) 2*length of LT line, (ii) 2*length of RT line, and (iii) the length of the RS line.
Otherwise
(b) (s)he chooses the RS line if the length of the RS line is less than (i) 2*length of LT line, (ii) 2*length of RT line, and (iii) less than or equal to the length of the LS line.
Otherwise,
(c) (s)he chooses the LT line if the length of the LT line is ≤ the length of the RT line; otherwise, (s)he chooses the RT line.
Procedure to load to the lift: LT and RT lines have the priority (over LS/RS lines), and alternate to load when they are both nonempty. When one, say LT, is empty and the other (RT) is not, RT skiers keep loading. However, since the Yodeler chair is a quad chair, for the fourth seat of each chair, LS and RS lines alternate when both are nonempty. Again, when one, say LS, is empty and RS is not, RS skiers keep loading to the fourth seat of the quad chair.
Assume that we have a ski-lift operator (a daemon process) that continuously checks the loading lines, and, makes sure that skiers follow the loading procedure. We do not have an upper bound on the length of each line, i.e., there may be an unlimited number of skiers waiting to load at a time. Develop a concurrent algorithm to implement this synchronization.

The Printer Daemon Problem. Consider an OS environment where processes have priorities. The printer daemon (a service routine) controls the allocation of three identical laser printers directly to processes: processes *GetPrinter*, use it for a while to print, and then *ReleasePrinter*. Write a *monitor* (as the printer daemon) where, if two processes execute *GetPrinter* and wait until a printer becomes available then, regardless of the order of their *GetPrinter* procedure call, the one with the higher priority gets the printer.
Assume that

- for condition variables, you do not have the conditional wait primitive; you can only use the basic wait and signal primitives for condition variables.
- All processes have different priorities.

- The monitor does not know the number (or the maximum number) of processes that exists within the system at any given time.
- The monitor does not use lists that dynamically increase in size (because they may become arbitrarily large and create performance problems).

The Four-of-a-Kind Problem. Develop a solution to the four-of-a-kind problem: There is a deck of 24 cards, split into 6 different kinds, 4 cards of each kind. There are 4 players (processes); each player can hold 4 cards. Between each pair of adjacent (i.e., seated next to each other) players, there is a (possibly empty) pile of cards. Each player behaves according to the following program.

```
While ((hand does not contain four of a kind) and (no one has won))
    begin
        Discard a card into the left-hand pile;
        Pick up a card from the right-hand pile;
    endwhile;
 if hand contains four of a kind then claim victory;
```

There are no ties; when a player has claimed victory, all other players stop. The game begins by dealing four cards to each player and putting two cards on the pile between each pair of adjacent players.

The Sleeping Barber Problem. A barbershop has a single room with one barber chair and n customer chairs. Customers (i.e., independent processes) enter the waiting room one at a time if empty chairs are available, otherwise they go elsewhere. Each time the barber (i.e., an independent process) finishes a haircut the customer leaves to go elsewhere, and, among the waiting customers, the one who has been waiting the longest (i.e., a FIFO waiting queue) moves to the barber's chair to get a haircut. If the barber discovers that the waiting room is empty, (s)he falls asleep in the waiting room. An arriving customer finding the barber asleep wakes the barber (i.e., a rendezvous takes place) and has a haircut; otherwise the arriving customer waits. Using semaphores and shared variables only (but not critical region statements or monitors), give two algorithms, one for the barber, and another for an arbitrary customer, that implement the sleeping barber problem.

The Dining Savages Problem. A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot. Develop a concurrent algorithm for the actions of the savages and the cook.

The Sleeping Barber Problem With Multiple Barbers: Revise the above problem into the following version (this is a real example from my barbershop): There are two barbers (Tina and Judy) and two barber chairs. Customers entering the shop declare three preferences: Tina, Judy or "don't-care". Waiting customers will still be serviced FIFO (i.e., the system satisfies the fairness and the liveness properties) with the exception of (a) those waiting for Tina will decline when Judy is available and wait for Tina in a FIFO order, and (b) those waiting for Judy will decline when Tina is available and wait for Judy in a FIFO order. Using a concurrent algorithm, give five algorithms, one for each barber, and three for each customer type, that implement the sleeping barber problem with two barbers.

File Buffer Allocation Problem. Many operating systems, such as UNIX, maintain a cache of file access buffers. each buffer is the size of a disk block. When a user process wants to read a disk block, the file system first looks in the cache. If the block is there, the file system returns the data to the user. Otherwise, the file system selects the least recently used buffer, reads the disk block into it, then returns the data to the user.

Similarly, if a user process wants to write a disk block that is in the cache, the file system simply updates the block. Otherwise, the file system selects the least recently used buffer and writes into that one. The file system keeps track of which buffers contain new data (i.e., which have been modified) and writes them to disk before letting them be used for a different disk block. (This is called a write-back cache policy). Develop a concurrent algorithm to implement a buffer cache having the above specifications.

The Direct Printer Allocation Protocol. Consider an OS environment where processes have priorities and get directly allocated printers when they want to print. There are n printers, and each printer has two states: *Available,* and *InUse.* If all n printers are *InUse* then processes wanting to print are blocked. When a printer's status changes from *InUse* to *Available* then, among those processes waiting to print, the one with the highest priority wins (not the one that was the first that wanted to print).

*P*rocesses wanting to print do so as follows. They get a printer using *GetPrinter()*, print using procedure *Print()*, and release the printer using *ReleasePrinter()*. The printer allocation protocol implements the following rules:

- *GetPrinter()* and/or *ReleasePrinter()* are executed mutually exclusively.
- Once a printer is given to a process, it is a dedicated resource for that process until release.
- Assume process *P* decides to use a printer to print. *P* executes *GetPrinter(),* which performs the following tasks:
    - Check all the printers, and go to sleep if none is available. Upon awakening, find out which printer is allocated to *P*, and return the id of the allocated printer.
    - Assume the printer R_i is available. Then, change the state of R_i from *Available* to *InUse,* and return the id i of the allocated printer.
- Once a printer R_i is allocated to *P*, *P* issues possibly multiple *Print*(*DataToBePrinted,* R_i) commands, and, eventually calls *ReleasePrinter()* to release R_i.
- *ReleasePrinter()* performs the following tasks: Check if there are processes waiting to use a printer, and if there are none then change the status of R_i as *Available.* If there are processes waiting to print, find the highest priority process waiting to print, say P_j, allocate R_i to P_j; release P_j.

Write a concurrent algorithm to solve the printer allocation problem by implementing *GetPrinter()* and *ReleasePrinter()* using semaphores.

The Meeting Problem. Two kinds of processes, A's and B's, enter a room and "meet" processes of other type. For a monitor-based solution, the definition of "enters' is: the process is allowed into the monitor (i.e., the room). The definition of "meets" we use is: Once a process of one type enters the room, it is considered to have met all the processes of the other type that are in the room. (Another possible definition is: a process of one type enters the room, and starts shaking hands with processes of the other type (in the order of their arrival). Handshaking constitutes an instance of "A meets B". We will not use this definition).

An A process cannot leave until it meets two B processes, and a B process cannot leave until it meets one A process. A process (of type A or B) leaves the room once it has met the required number of the other kind of processes. Develop a concurrent algorithm to implement this synchronization.

Modify your answer to the above part so that the first of the two B processes that meet an A process does not leave the room until after the A process meets a second B process. Develop a concurrent algorithm to solve the Meeting Problem.

The UniSex Bathroom Problem. Suppose there is a bathroom with n stalls (a compartment with a toilet, or, as the British say, "a water closet") in the Thwing Center. The bathroom can be used by both men and women, but not at the same time. Give a monitor-based concurrent algorithm that controls the use of the bathroom. The bathroom entry is by the monitor procedure *EnterBathroom*. After entering the bathroom successfully, individuals use the monitor procedure *getstall* to get to a stall. If all stalls are in use then they wait (there is enough space in the bathroom). After using a stall, each individual calls *ReleaseStall* for others to use it.

The monitor is fair in the following sense. Assume that, at a given point in time, the bathroom is in use by x individuals of one sex (some using stalls and some waiting), and the first individual, say P, from the opposite sex arrives. Then

- P enters the bathroom immediately after the bathroom exit of those x individuals,
- when P is waiting, if other individuals of the same sex arrive, they use the bathroom concurrently with P,
- when P is waiting, if individuals of the opposite sex arrive for bathroom use, they enter the bathroom after P (and the company, if any) exit,
- when P (and the company) is using the bathroom, if individuals of the same sex arrive then they wait until those of the opposite sex with P and waiting get to use the bathroom.