Program 5: Saturday Morning Chores / Threads

CSCI 4547 / 6647 Systems Programming
Spring 2022

1 Sharing the Chores

Assignments 5 and 6 will both implement a game using concurrency, but the concurrency will be implemented in two very different ways.

- P5. A process with four subsidiary threads. All four and the parent process (”Mom”) interact through shared memory.
- P6. Mom will be server process that interacts with four client processes (kids) through sockets. All five interact through a protocol.

1.1 Goals for P5

1. To simulate a concurrent process (doing the Saturday morning chores).
2. To implement application logic for, and concurrently operate, Mom and the Kids.
3. To implement a shared-memory class, with synchronization.
4. To use signals and a signal handler.

2 Scenario

The family. Mom has four children, Ali, Cory, Lee, and Pat. She also has an unending list of chores that need to be done every week. On Saturday, Mom posts the job list and everybody knows that Mom will make them keep doing chores until she signals the end of the work session. They do not know that Mom plans to make them work approximately 3.5 hours (21 time units).

Varied jobs. There are many kinds of jobs. Mom rates each one on three scales (1..5):

1. Q: From quick to time consuming. A quick job = 1 time unit; a time consuming job = 5. Each time unit is 10 minutes,
2. P: From pleasant =1 to very dirty, unpleasant, or sweaty = 5
3. E: From easy work = 1 to heavy work = 5

Mom rewards each worker-child with points for every completed job. The number of points is: Q*(P+E). For example, Cleaning the toilet = quick, somewhat unpleasant, light work = 1*(4+1) = 5 points. Mowing the lawn is slow, fairly sweaty, and strenuous = 5*(4+5) = 45 points.

Mom has an infinite number of jobs of all sorts, which she posts as needed. However, no more than 10 are posted at any given time. On Saturday morning, early, Mom posts the initial set of jobs for the day. Each kid looks at the posted jobs and chooses one according to his/her moods and the jobs' properties. The kid then tags the chosen job with his/her name and later marks it as “finished”. Mom will then remove it from the job-board, put it in a vector of done jobs, and replace it in the job table by another job.

Kids are moody. From week to week, a child may have different moods, and that leads to different job-preferences. Sometimes children are *lazy* and won't do heavy work. Sometimes they are *prissy* and won't do dirty work. *Overtired* children like short jobs because they can't concentrate on a job for long. A child can feel *greedy* sometimes – and go for the biggest available reward. Finally, everybody is sometimes just *cooperative*and grabs the last job on the list. The kids select jobs according to their moods and the jobs' properties, do them, and report back to Mom.

3 Helper classes.

Enums. This application uses two sets of symbols; create and use an enumerated type for each.

- Moods (lazy, prissy, over tired, greedy, cooperative).
- Job status (not started, working, complete).

The Job class. This class definition is needed by both Mom and Kids. It will have the following data members:

- 3 short ints between 1 and 5 indicating how slow/dirty/heavy the job is,
- An int for the value of the job, calculated from the properties.
- A string for a kid's name.
- An enum field to represent the job status.

Job functions:

- A default constructor that initializes all data members appropriately. Use random numbers for the job properties.
- `chooseJob( string kidname, int jobNumber)`.
- `announceDone( )` A child will call `announceDone( )` to set the `done` field to true when the job is completed.

4 The Shared Data and Global Function.

The JobTable class. Mom will own a single instance of this class, and pass a pointer to each Kid thread when it is created. Into this class, put all the things that must be **shared**:

1. A table of 10 jobs. Mom will fill in the initial list of jobs.
2. A mutex lock to protect the table, initially unlocked. Remember that this structure must be locked and unlocked every time it is used.
3. A variable to coordinate all parties, quitFlag, initially false.

The kidMain() function must be a static global function. It is the 3rd parameter in the thread-creation command.

When a thread is created, this function will be called. At this stage of thread creation, everything is operating in what we call the "static world", which means "the part of the program that is outside the class structure". The first and only action of `kidMain()` must be to transfer control into the OO world. When the 4th parameter in the thread-creation command parameter is cast back to a Kid*, it can be used to call a normal function in the Kid class. I will call this function `Kid::run()`; it is essentially the Kid's main program. After that, all operations are performed in the OO world.

5 The Mom class

This is the application controller. Data parts are:

1. An instance of the JobTable.
2. A constant array of 4 Kid names.
3. An array of the tids of the 4 Kid-threads.
4. A vector to store the completed jobs.
5. A `time_t` variable to store the time at which the chores started and another to store the current time.

Function members.

1. A default Mom constructor.
2. A function that initializes the job table. Lock the table first, then call the Job constructor in a loop to fill it with jobs.
3. A function for scanning the job table looking for a finished job. When found, move the Job object to the vector of completed jobs, then replace it with a newly created Job.
4. A `print()` function.
5. Any other functions that you need, but make them private.

Mom's run function.

1. Initialize the job table.
2. Instantiate Kid `k` using the kth name in the array of kidNames and a pointer to the JobTable object. Store the Kid in Mom's array of Kids.
3. Create a thread for the kid and store its tid in the array of tids. The parameter for thread creation should be a pointer to the current Kid. Have the thread execute a global static function named doKid(), described below.

As soon as the 4th kid is running, store the current time in your `time_t` variable and send a start signal to each of the kids by using `pthread_kill(tid, SIGUSR1)`. Then:

1. Enter a loop.
2. Sleep for one second.
3. When the timer signal wakes you up, look at the clock and calculate the difference between the current time and the saved time. If it is 21 or more, leave the loop.
4. Otherwise, scan the job table, save the finished jobs, and replace them in the table by new jobs.
5. When you leave the loop, send four QUIT signals, one to each of the four kids. Wait for them to join you.

6. When all four kids have joined, use the vector of completed jobs to compute how much each kid has earned. Award an extra $5 to the kid with the highest earnings. Print the finished-job list and the winner's name If all is working correctly, Mom's computations will agree with the Kids own computations.

6 The Kid class.

A Kid's Data Members.
Data members:

- The Kid's name.
- The Kid's mood (an enum code).
- A vector to store the Kid's completed jobs.

Kids' Functions.

1. A constructor with two parameters, the Kid's name and a JobTable*. Create an empty signal set, add SIGUSR1 and SIGQUIT, and attach it to the thread by calling `pthread_sigmask()`.
2. A `print()` function, of course.
3. A function for choosing and saving the mood of the day.
4. A function for selecting a job that fits today's mood. Avoid selecting a job that has already been started but is not yet finished. When you select a job, put your name in its name field and return.
5. **Kid::run()**, explained below, which will be called by kidMain when a new Kid thread is create.

The Kid::run() function

- First create a signal-processing apparatus for the two signals that Mom will send to Kids. You will need an empty signal set to which you add `SIGUSR1` and `SIGQUIT`. Set up and initialize a sigAct object and attach it to the thread with a call on `sigaction()`. In your sigaction code, figure out which signal was sent, and set one of the shared flags to true. This flag will be seen by all threads.
- Select and save a mood from the list of five moods (lazy, prissy, overtired, greedy, cooperative). This will be used in a switch to determine which job-selection process is used. Implement a function for each of these selection methods:
  - A lazy child chooses the easiest job.
  - An overtired child chooses the shortest job.
  - A prissy child chooses the cleanest job.
  - A greedy child chooses the biggest payoff.
  - A cooperative child doesn't care which job he/she gets, so always takes job 9.
- Then wait for a signal. When you are waked up by a `SIGUSR1`, continue thus:
  - Select a job and call the function for today's mood.
  - Simulate doing the job by sleeping for the required number of seconds.

  - When the nap time is over, check the `quitFlag`. If it has been set to true, leave the work loop.
  - If you wake up because the QUIT signal has been sent, set the the `quitFlag` to true and break out of the work loop.
  - Otherwise, move the current Job object to your vector of completed jobs and repeat the loop.

- When you break out of the work loop, print a message that you got the signal, then print the contents of your job vector and the amount of money you earned, then quit. Note that you will not be paid for the final partly-done job.

7 The main program.

Print the banner, instantiate Mom and call her run() function. Run this program several times and submit all the results, including the list of jobs done by the children, their final scores, and the winner. I will check to be sure that your trials are not all the same.

7.1 Testing

First write and compile the logic for Mom. When it compiles without warnings, write the Kid code. T test Mom with one child and verify that the thread works. Then test with 2 kids, then with 4.

The test plan. Make a test plan that calls every function at least once to ensure that the functions all work and produce the expected results. DO NOT SKIP THIS STEP. Run the program repeatedly and make sure that the output is not always the same. Turn in two sets of output.