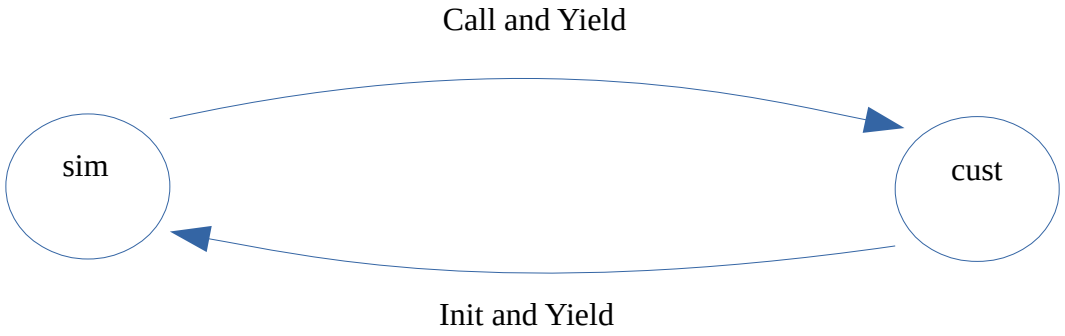
David Perry **CS543 HW#5 Report** **December 8, 2014**

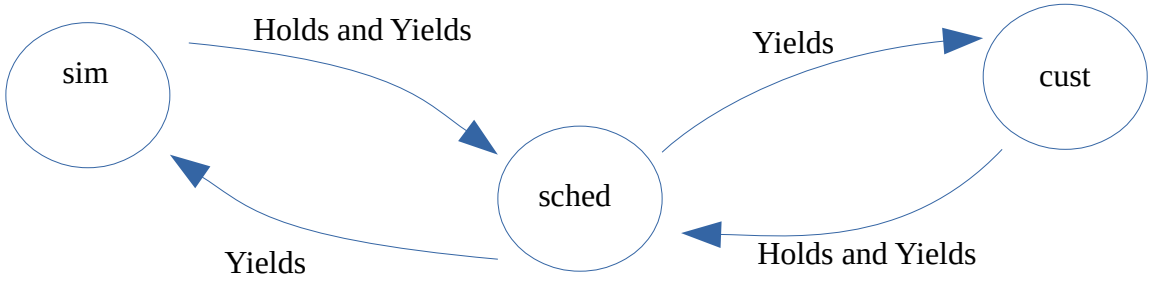
This report will briefly describe what a threads-based discrete-event simulator is, discuss implementation details involved with CSIM, and give an overview of my own simulation library implementation.

Sim to Cust:
The first step to understanding how the simulator works is to understand what it is attempting to do. In a threads-based discrete-event simulator actions are modeled as threads. These actions are written as functions by a user wishing to run a simulation. Each action is launched as its own thread and performs some meaningful task that is being simulated. Meaningful tasks include waiting for other events to complete, holding for a certain amount of time, and reserving/releasing facilities. Most importantly, the simulation can record statistics as the simulation occurs creating data that can be used for practical or experimental purposes. The simulation also has a scheduler responsible for ensuring that actions are executed in the correct order which ensures a correct simulation.

The most obvious challenge with implementing a threads-based discrete-event simulator is the synchronization of various threads. The first challenge in doing so is handling the first case of control transfer between “sim” and “cust”. To solve this issue when sim calls the function for cust it also has to yield to that process. This allows cust to run and initialize it's state. After initialization is over cust yields control back to sim. For example:



After handling this case the simulation must ensure that only one thread runs at a time. To handle this issue CSIM uses the special “scheduler” thread and uses various thread functions to enforce this policy. Essentially, every time a thread realizes it is done executing (finishes or blocks) it yields control back to the scheduler thread. Once the scheduler has control the most recently time-stamped activation record (more explanation later) is pulled from a calendar. Finally, the scheduler uses this activation record to determine the correct thread to run and yields, giving control to the proper thread. For example:



Hold:
Another important function that CSIM implements is the hold operation. When a thread calls the hold function it wishes to advance the clock forward in the simulation by some amount. Typically, this is used to simulate some amount of time passing during some functionality (ie reserving a facility). However, the hold function is complicated by the fact that there can be other threads that need to be ran before the time advances to ensure correct order of execution.

To handle this issue CSIM does a few things in its hold function. One of the first things it must do is create an activation record with a time-stamp set for what time it is now plus the amount of time that the thread should hold. This activation record is then placed in the simulation calendar. Finally, the thread yields control to the scheduler, the scheduler pulls the most recent activation record from the calendar, and yields control to the corresponding thread.

Activation Record:
In my simulation I choose to create a C++ class to represent my Activation Record. This class contained the members thread_id, and thread_control_block. The thread_id corresponds to the pthread_t variable needed for creating processes using pthreads. The thread_control_block is another class I wrote and contains a pthread_cond_t variable which is what a process waits when it has finished executing (termination or blocks) and signals when it wants to relinquish CPU to another process.

Create:
I took a different approach to CSIM when designing my create function but similarly to CSIM I created my own create function. When called this function uses pthread_create to create a new thread that executes a function I wrote called dispatchFunc. This function does some initialization including the creation of the conditional variable used for synchronization, creates and inserts an activation record into the calendar, and adds the processes Thread Control Block to the thread_map (map used for looking up thread details). After this initialization the actual simulation function is ran (I used a function pointer as a parameter to dispatchFunc). Finally, after the function has completed running dispatchFunc removes the thread from the thread_map and signals the next thread to run.