3bb4 Assignment 3

Peter Hoang 1208271
Tim Li 1205961
Seungwoo Cho 1205098
Mitchell Spector 1208006

There were five properties introduced at the beginning of the project we must keep and here are the explanations for the upholding of these properties

(a) Infinitely many processes are generated.

The **GENERATOR** class exhibits this behaviour specifically, when the generator class produces more **PROCESSES** than can fit in the **QUEUE** the **GENERATOR** will loop on its last state unable to add anymore **PROCESSES** onto the **QUEUE**.

(b) The whole system is deadlock free.

Recall the four deadlock condition are:

1. Serially reusable resources - shared resources under mutual exclusion
2. Incremental acquisition - the holding on of acquired resources allocated, while waiting for additional resources
3. No preemption - resources cannot be forcibly withdrawn
4. Wait-for cycle - Circular chain of processes waiting on each other

The **GENERATOR** creates unique processes that are not reusable once the processes finish their tasks (destroyed by **GRIMREAPER**), meaning that there are no serially reusable resources. In addition, when the **CPU** returns processes they are not shared and simply used by one **CPU**. As long as one of the conditions are to be false, it is safe to say the system is deadlock free.

In addition, the **CPU** does not hold onto processes if they complete or not finish, the task will either be placed back into the **QUEUE** or simply terminated by the **CPU** preventing resources (processes from being held) and **CPU** only runs one process at a time not waiting for any process.

Finally since no resources are being held preemption and wait for cycles cannot occur

(c) No process with remaining execution time is destroyed.

This is done by checking the time it takes to finish specific tasks. Our system will have two different states, which will be determined by a task's time it takes to complete the task. If a task is not done within the time limit, the task will be stored back to the **QUEUE** by notdone_task action, preventing the action from destroyed without the completion. In contrast, when the process is done within the time limit, done_task action will pass the process to **GRIMREAPER**, which will destroy finished tasks.

(d) No process uses the CPU more that its allowed time.
The transition notdone_task and done_task for the **GRIMREAPER/CPU** indicates this behaviour, when the CPU **countdown** runs out and a process is incomplete then the process is placed back into queue. Otherwise if the process completes within the specified amount of time then the process is simply destroyed by the GRIMREAPER.

(e) No process with no-remaining time is scheduled for execution.
The transition done_task passes the processes with no-remaining time to the **GRIMREAPER,** which will destroy the process from the queue. Therefore, if a process has no-remaining time, it is impossible to be scheduled for execution.

Details regarding each individual Component

GENERATOR *-Peter Hoang*

GENERATOR shows the number of processes that can be generated and then loaded onto the QUEUE, after our fixed number of processes that can be loaded onto the QUEUE in our case two. The GENERATOR will still continually producing processes, this is indicated by the loop on itself.

Transitions
generate
enqueue
enqueueOverflow - The GENERATOR continues to produce more processes these are not enqueued however.

States
GENERATOR[1...2] //max number of processes is two
//GENERATOR[2] is start state counting down the number that of processes that we can still generate and enqueue

QUEUE - *Mitchell Spector*

The **QUEUE** interpretation is a simple one as indicated by the discussion boards, the interpretation is not the possible combinations that can exist within the queue. But instead the number of items within the **QUEUE**. With the assumption that the items first placed into the queue are taken out first. Furthermore, the **QUEUE** has an error states, that will track overflow of actions enqueue. Note that we modified the PORT example, from page 16 of the chapter 10 lecture slide, to be suitable for **QUEUE** in our system. The assumption is made that the LTSA composition hides the index of individual components in enqueue/dequeue, and shows all possible enqueue/dequeue combinations. The overflow enqueue is indicated by the error state.

DISPATCHER -Seungwoo Cho

DISPATCHER works as a bridge(loading processes) between **QUEUE** and **CPU**. **DISPATCHER** sends ordered processes to be completed to **CPU** by *start* and *dequeue* action. Note that the process shares the same transition name with the **QUEUE** and the same transition start state as the **CPU**. The assumption is made that the start and dequeue are executed simultaneously.
Behaves similarly to the employee who loads the vending machine on the midterm, simply shares the same methods as the queue and the **GRIMREAPER**.

transitions

start
dequeue

States

DISPATCHER

CPU - *Tim Li*

CPU has a fixed Quantum time in which a fetched process from the **DISPATCHER** will run. The **CPU** will tick until the quantum time is up, in which the process can be done at anytime and the **CPU** restarts for the next process after their respective deaths. If the quantum is finished but the process is not done, the dispatcher is notified, and the **CPU** restarts for the next process. The assumption is made that when the process is returned to the QUEUE, the time left to finish the process (the difference beyond the quantum) is not tracked by states.

transitions
dequeue
done_task
States
CPU[0..3]

GRIMREAPER - *Tim Li*

GRIMREAPER destroys finished processes. Otherwise, if they exceed quantum time, put process back into queue. Both return GRIMREAPER to ready state to receive further tasks from CPU.

RR_SCHEDULER - everyone

Everything combined together with the composition symbol.

The Code Itself

```
const PROCESSES = 2 // Total number of PROCESSES
range T=0..PROCESSES

GENERATOR = GENERATOR[2], //start state is 4 PROCESSES LEFT TO GENERATE
GENERATOR[i:T] =
if(i>0) then
        ( generate->enqueue->GENERATOR[i-1]) // REDUCE THE AMOUNT GENERATOR
CAN GENERATE
else (enqueueOverflow ->GENERATOR[i]).

const N=2 /*2 processes max enqueued*/
range P = 0..N

range M = 0..1
set   S = {[M],[M]} //changes for two spots


QUEUE            //empty state, only send permitted
  = (enqueue[x:M]->QUEUE[x]),
QUEUE[h:M]       //one message queued to QUEUE
  = (enqueue[x:M]->QUEUE[x][h]
    |dequeue[h]->QUEUE
    ),
QUEUE[t:S][h:M]  //two messages sent to QUEUE
   = (enqueue[x:M]->QUEUE[x][t][h] // three states indicate overflow this leads to error
detection
     |dequeue[h]->QUEUE[t]
     ).
||AQUEUE = QUEUE/{enqueue/enqueue[M],dequeue/dequeue[M]}.


DISPATCHER = (start-> dequeue ->DISPATCHER). // ASSUMPTION THAT start and
dequeue occur at the same time

const QUANTUM = 3 /*3 quantum units defined arbitrarily*/ // The max amount of time a
process can run

CPU(N=QUANTUM) = (start->CPU[N]), //similar to COUNTDOWN code in chapter two
CPU[i:0..N] =
(when(i>0) working->CPU[i-1]|done_task -> death ->CPU[QUANTUM] /*CPU will "work" until
done or when quantum is finished*/
|when(i==0) notdone_task->CPU[QUANTUM] /*if quantum finished, indicate over_time for
GRIMREAPER*/
).

GRIMREAPER = (done_task-> death -> GRIMREAPER| notdone_task -> enqueue ->
GRIMREAPER).
/*destroy_thread using process ID, return to GRIMREAPER ready state*/

||RR_SCHEDULER = (GENERATOR || AQUEUE || DISPATCHER || CPU || GRIMREAPER).
// combination of all components together
```