COM304: Project 2
Santa's Workshop

In this project, the probability distributions are done by creating nextGiftType() method. In this method,

- 0's indicate that no gift request is arrived and this has a probability of %10
- 1's indicate type 1 gift requests are arrived with a probability of %40
- 2's indicate type 2 gift requests are arrived with a probability of %20
- 3's indicate type 3 gift requests are arrived with a probability of %20
- 4's indicate type 4 gift requests are arrived with a probability of %5
- 5's indicate type 5 gift requests are arrived with a probability of %5

```
int types[20] = {0,0,1,1,1,1,1,1,1,1,2,2,2,2,3,3,3,3,4,5};
        int index = rand() % 20;
        return types[index];
```

Figure 1

Different tasks are implemented in separate methods for demonstration of acquiring the lock, dequeuing and releasing the lock clearly. As an illustration, PackagingTask(void *arg) method is implemented as follows:

```
void* PackagingTask(void *arg){
    if (!isEmpty(packaging_queue)&& current_task_ID == packaging_queue-
>head->data.ID){
            printf("elf in if\n");
            pthread_mutex_lock(&packaging_mut);
            Task ret = Dequeue(packaging_queue);
            ret.responsible = "A";  //TODO
            pthread_sleep(1); //packaging time 1 sec
            printf("dequeued: %d\n", ret.ID);
            pthread_mutex_unlock(&packaging_mut);
            printLog(&ret);
            addDeliveryQueue(ret.giftType, ret.giftID);
            incrementCurrentTask();

    }
}
```

Figure 2

If the packaging queue is not empty, and the task is at the head of the queue, packaging mutex acquires the lock and the corresponding task is dequeued from packaging queue. Since packaging time requires 1 second, we used pthread_sleep(1) and then packaging mutex releases the lock.
Then, printLog method is called to print the values of this task into events.log file. The packaging task is then send to addDeliveryQueue method and the current task ID is incremented by one.

What printLog method does is it acquires the lock using log_mut pthread_mutex and then events.log file is opened for printing the TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT and Responsible.

```
void printLog(Task* t){
    printf("print CALLED\n");
    pthread_mutex_lock(&log_mut);
    FILE *f = fopen("events.log", "a");
    fprintf(f, "%d\t\t%d\t\t%d\t\t\t%s\t\t\t%d\t\t\t%s\n", t->ID, t-
>giftID, t->giftType, t->type, t->requestTime, t->responsible);
    pthread_mutex_unlock(&log_mut);
    fclose(f);
}
```

Figure 3

After creating the tasks, ElfA is assigned to packaging task and painting task; ElfB is assigned to packaging task and assembly task; and Santa is assigned to delivery task and QA task as instructed.

```
// manages ElfA's tasks.
void* ElfA(void *arg){
    while(passedTime()<simulationTime){
        //pthread_sleep(1); //TODO
        PackagingTask(NULL);
        PaintingTask(NULL); //only does paint

    }
    pthread_exit(0);
}

// manages ElfB's tasks.
void* ElfB(void *arg){
```

```
    while(passedTime()<simulationTime){
        //pthread_sleep(1); //TODO
        PackagingTask(NULL);
        AssemblyTask(NULL); //only does assembly

    }
    pthread_exit(0);
}

// manages Santa's tasks
void* Santa(void *arg){
    while(passedTime()<simulationTime){
        //pthread_sleep(1); //TODO
        DeliveryTask(NULL); //prioritizes delivery
        QATask(NULL);

    }
    pthread_exit(0);

}
```

Figure 4

The threads for elfA, elfB and Santa and the related queues and output are controlled using ControlThread function as shown in *Figure 5*.

```
void* ControlThread(void *arg){
    pthread_t elfAThread, elfBThread, santaThread;

    pthread_create(&elfAThread, NULL, ElfA, NULL);
    pthread_create(&elfBThread, NULL, ElfB, NULL);
    pthread_create(&santaThread, NULL, Santa, NULL);

    pthread_detach(elfBThread);
    pthread_detach(elfAThread);
    pthread_detach(santaThread);
    pthread_exit(0);
}
```

Figure 5

By using the probabilities that are indicated in Figure 1, gift type is determined and according to those gift types, pthreads are created. In pthread_create, specific function for adding to corresponding queue is used.

```
if(gift_type == 1){
                pthread_t type1_thread;
                pthread_create(&type1_thread, NULL, doType1, giftT);
            }
```

Figure 6

As an example, *Figure 6* illustrates the implementation for gift_type==1. For this gift type, packaging is handled by putting &type1_thread as thread, NULL as attributes, doType1 as a corresponding function and gift type as an argument into pthread_create.

Inside doType1, package thread is created using corresponding functions. For instance for creating package thread we used addPackageQueueType1 function. Then pthread_join is used.

```
void* doType1(void *arg){
    void *status;
    //int giftT = *((int *) arg);
    pthread_t package_thread1, delivery_thread1;
    printf("package + deliver only\n");

    pthread_create(&package_thread1, NULL, addPackageQueueType1, arg);
    pthread_join(packaging_thread, NULL);
}
```

Figure 7

In addPackageQueueType1 function, createTask function is called for creating and assigning values into a specific task (see Figure 8).

In createTask method the task is created and it’s ID, type, giftType, giftID, and requestTime are assigned.

```
Task* createTask(int giftType, char* taskType){
    Task *t = (Task *) malloc(sizeof(Task));
    pthread_mutex_lock(&taskCount_mut);
        task_count++;
        printf("task count:  %d\n", task_count);
        pthread_mutex_unlock(&taskCount_mut);
        t->ID = task_count;
        t->type = taskType; //TODO
        t->giftType = giftType;
```

```
        t->requestTime = (int) passedTime();
        pthread_mutex_lock(&gift_count_mut);
        t->giftID = gift_count;
        pthread_mutex_unlock(&gift_count_mut);
        return t;
}
```

Figure 8

Then packaging_mut pthread_mutex acquires the lock and the task is added to packaging_queue using Enqueue(packaging_queue, *t). Finally, the packaging_mut releases the lock.

```
void* addPackageQueueType1(void *arg){
    int giftType = *((int *) arg);
    Task* t = createTask(giftType, "C");
    pthread_mutex_lock(&packaging_mut);
    Enqueue(packaging_queue, *t);
    pthread_mutex_unlock(&packaging_mut);
    pthread_exit(0);
    }
```

Figure 9

The other methods for adding tasks to specific queues such as addDeliveryQueue, addPaintingQueue are implemented in a similar manner.
For gift type 1, 2 and 3 gifts the procedure goes in a similar way. However, for type 4 and 5 gifts the procedure is slightly different since for type 4 Painting and QA tasks run in parallel and for type 5 Assembly and QA run in parallel. In order to make sure that tasks for those type of gifts run in parallel and to proceed when both parallel tasks are finished, incrementType4CondCounter(Task t) is used for type 4 and incrementType5CondCounter(Task t) is used for type 5.

```
void incrementType4CondCounter(Task t){
    if (t.giftType == 4){
        pthread_mutex_lock(&type4_package_cond_mut);
        type4_package_cond++;
        pthread_mutex_unlock(&type4_package_cond_mut);
    }
}
```

Figure 10

Figure 9 illustrates incrementType4CondCounter (Task t) method where type4_package_cond is incremented by one. This method is

called in PaintingTask and QA task methods since for type 4 gifts Painting and QA runs in parallel.

type4_package_cond is important to determine the tasks are running in parallel are both finished executing. When this condition integer is divisible by 2 and the integer itself is not 0, this means that both tasks are done, hence we are leaving the while loop using "break". (*see Figure 11*)

```
while (ret.giftType == 4){
                pthread_mutex_lock(&type4_package_cond_mut);
                int now_type4_cond = type4_package_cond;
                //printf("waits for type4 finish.. %d\n",
type4_package_cond);

                //pthread_sleep(1);
                pthread_mutex_unlock(&type4_package_cond_mut);
                if ((now_type4_cond % 2) == 0 && now_type4_cond != 0)
break;

            }
```

Figure 11

Additionally, createDownTask(int giftType, int giftID, char* taskType) method is constructed for creating downstream task to hold giftID steady. (*see Figure 12*)

```
Task* createDownTask(int giftType, int giftID, char* taskType){
    Task *t = (Task *) malloc(sizeof(Task));
    pthread_mutex_lock(&taskCount_mut);
        task_count++;
        printf("task count:  %d\n", task_count);
        pthread_mutex_unlock(&taskCount_mut);
        t->ID = task_count;
        t->type = taskType; //TODO
        t->giftType = giftType;
        t->requestTime = (int) passedTime();

        t->giftID = giftID;

        return t;
}
```

Figure 12

Part 2

For this part since there is no more presents are waiting to be delivered, 0's in the type array is replaced with 1's to make sure that gifts of type 1 will have %50 chance instead of %40.

```
/* This is for probability demonstration*/
int nextGiftType(){
        int types[20] = {1,1,1,1,1,1,1,1,1,1,2,2,2,2,3,3,3,3,4,5};
        int index = rand() % 20;
        //printf("index: %d %d",index, types[index]);
        return types[index];
}
```

Figure 13

To sustain 3 or more GameStations are waiting to go through QA, we used XOR to show that when size of the QA queue is more than 3 it must be not be empty for QA task to proceed.

```
void* Santa(void *arg){
    while(passedTime()<simulationTime){
        //pthread_sleep(1); //TODO
        //printf("Santa\n");
        DeliveryTask(NULL); //prioritizes delivery
        if (isEmpty(delivery_queue) != QA_queue->size>=3){  //XOR
            QATask(NULL);
        }
    }
    pthread_exit(0);

}
```

Part 3

Since every 30 second there is a gift request arriving from New Zealand. When the passed time is divisible by 30, pthread is created and type1_thread, doType1 method and gift type is put into it. Then gift_count_mut acquires the lock, gift_count is incremented by one and the mutex releases the lock.

```
//PART3 CONDITION ADDED
            if ((((int) passedTime())%30) == 0){
                printf("NEW ZEALAND\n");
                pthread_t type1_thread;
                pthread_create(&type1_thread, NULL, doType1, giftT);
                pthread_mutex_lock(&gift_count_mut);
                gift_count++;
                pthread_mutex_unlock(&gift_count_mut);

            }
```

Keeping Logs

For Part1;
events.log file:

TaskID	GiftID	GiftType	TaskType	RequestTime	Responsible
1	1	3	A	1	B
2	2	1	C	2	A
3	3	4	P	3	A
4	3	4	Q	3	S
5	1	3	C	3	A
6	4	1	C	4	A
7	2	1	D	4	S
8	5	4	Q	5	S
9	5	4	P	5	A

For Part 2;
events_part2.log file:

TaskID	GiftID	GiftType	TaskType	RequestTime	Responsible
1	3	A	1		B
2	1	C	2		A
3	4	Q	3		S
4	4	P	3		A
5	3	C	3		A
6	1	C	4		A
7	1	D	4		S

For Part 3;
events_part3.log file:

TaskID	GiftID	GiftType	TaskType	RequestTime	Responsible
1	3	A	2		B
2	1	C	2		A
3	4	P	3		A
4	4	Q	3		S
5	3	C	4		A
6	1	C	5		A
7	1	D	5		S