Final Report

구매 관리 시스템

과 목	임베디드시스템S/W설계
담당교수	김태석 교수님
학 과	컴퓨터공학과
학 번	2019202050
성 명	이강현
날 짜	2024. 06. 02 (일요일)

광운대학교
KwangWoon University

1. 요약

Kernel function 수	사용한 Task의 수	Semaphore 예제 수	Message queue 예제 수
3	5	5	8

먼저 kernel function은 총 3개로 OS_TASK.C 파일내에 작성하였다.

INT8U MyKernelFunc(INT8U prio, char *buffer) 함수는 우선순위를 입력받고 이를 해당하는 요청 문자열로 변환하고 추가로 입력받은 세부 내용을 저장하는 함수이다.

INT8U MyKernelPrintFunc() 함수는 프로그램이 종료될 때 저장된 내용 모두를 출력하는 함수이다.

INT8U MyKernelResponseFunc(INT8U response) 함수는 입력받은 인자에 따라 성공 여부를 저장하는 함수이다.

다음은 application내의 task들과 함수에 대한 설명이다.

void TaskInput(void* p_arg)는 사용자에게 입력을 받는 task이다.

void TaskUI(void* p_arg)는 사용자에게 내용을 보여주는 task이다.

void TaskLoadProducts(void* p_arg)는 상품 목록을 파일에서 불러오는 task이다.

void TaskCartManagement(void* p_arg)는 사용자의 카트내역을 관리하는 task이다.

void TaskPaymentProcessing(void* p_arg)는 사용자의 결제를 담당하는 task이다.

int CheckPassword(const char* inputPassword)는 비밀번호의 일치여부를 반환하는 함수이다.

OS_EVENT* Sem는 물품의 동시접근을 제어하는 세마포어이다.
모든 task에서 물품 구조체에 접근하는 경우 사용하여 5번 사용하였다.

OS_EVENT* QueueMsg는 task간 IPC를 위한 메시지 큐이다.

void* QueueMem[MAX_QUEUE_SIZE]는 메시지들이 저장되는 공간이다. TaskInput task에서 각 command마다 메시지 큐를 보내므로 5번 사용되고 TaskUI,TaskCartManagement,TaskPaymentProcessing에서 각각 1번씩 수신하여 분기를 나누기 위해 사용된다. 따라서 8번 사용하였다.

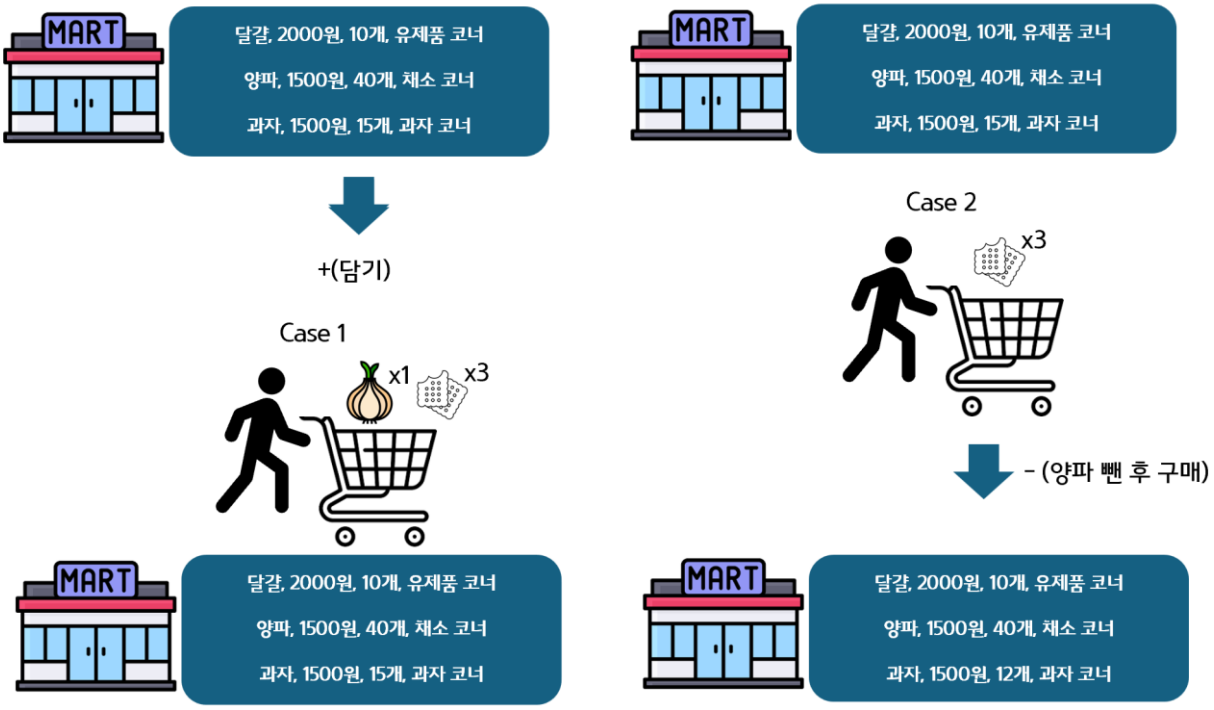
2. 본문

-프로젝트 개요-

구매 관리 시스템은 시장에 존재하는 물품들의 가격, 위치, 재고 정보를 제공하며 사용자의 요구에 따라 카트에 삽입, 삭제를 가능한 시스템이다. 추가적으로 카트에 존재하는 물품들을 비밀번호와 비교하여 구매할 수 있는 기능도 제공한다.

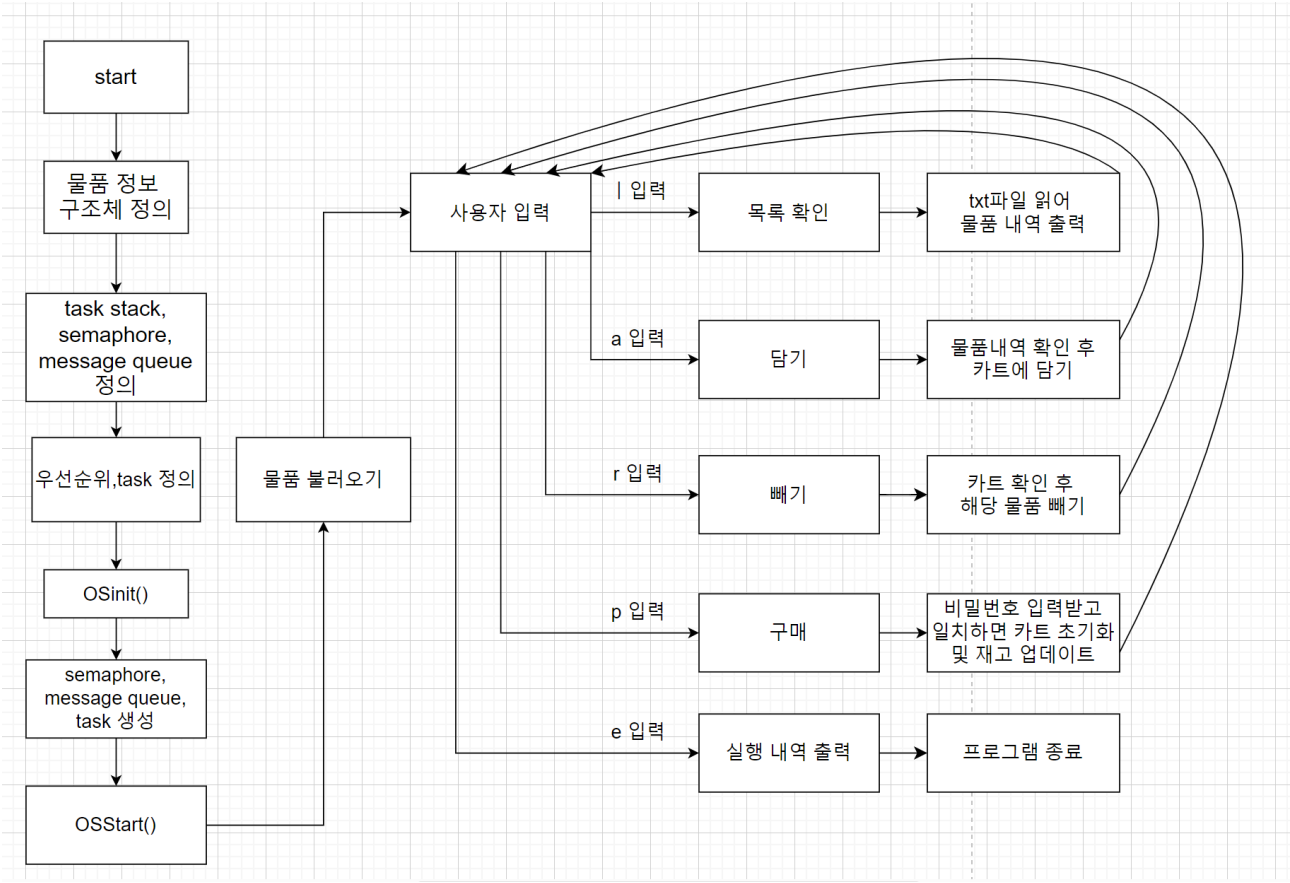
시장에 존재하는 물품들의 대한 정보를 txt파일로부터 로드하여 업데이트하고 사용자의 다양한 요구를 콘솔로 입력받아 물품목록, 담기, 구매등을 가능하게 할 예정이다. 또한 물품을 담거나 구매하여 물품 목록에 수정사항이 생겼을 때 이를 적절히 업데이트한다.

시스템의 전반적인 흐름이다.



case 1은 양파와 과자를 담은 상황이다. 하지만 재고에서 해당 물품의 개수가 줄어들지는 않는다. 그 다음 case 2는 양파를 빼고 상품을 구매하였다.
결과적으로 구매한 양파 3개만 재고에서 줄어들었다. 구매를 확정하기 전에는 전체 재고의 양은 줄지 않는다.

-프로젝트의 구조-



먼저 시작되면 필요한 구조체를 정의하고 task stack이나 semaphore message queue들을 정의한다.
Task 우선순위를 정하고 모든 task를 정의한다.
OSinit으로 초기화 후 semaphore, message queue, task를 생성하고 OSStart로 multitasking을 시작한다.
이후 task에 의해 물품을 파일에서 불러오고 사용자에 입력에 따라 실행되는 흐름이다.

-tasks-

Task들은 사용자의 입력을 받는 TaskInput, 물품을 불러오는 TaskLoadProducts를 제외하고 모두 실행되면 OSTaskSuspend로 스스로 CPU를 반납한다. 이후 TaskInput에서 실행되어야 하는 Task를 OSTaskResume를 이용해 실행가능하게 만든 후 스스로 딜레이를 걸어 CPU를 건네주는 방식으로 진행된다.

먼저 void TaskLoadProducts(void* p_arg)에 대한 설명이다.

우선순위는 10으로 가장 높으며 한번만 실행되는 초기화 task이다.
전체 물품 내용을 products.txt에서 ,를 구분자로 하여 불러오고 이를 구조체에 저장한다. 물품을 불러온 후 전체 물품내용을 출력하고 이후 사용자에게 사용할 수 있는 command를 출력해준 후 OSTaskDel(OS_PRIO_SELF)를 이용해 task를 스스로 종료한다.

다음은 void TaskInput(void* p_arg)에 대한 설명이다.

우선순위는 11로 한번만 실행되는 TaskLoadProducts를 제외하면 가장 높은 우선순위를 가진다.
_kbhit()을 이용하여 콘솔입력이 있는지 없는지 체크한 후 command에 맞게 분기하여 다음 task를 실행시키는 task이다. l과c을 입력한 경우 TaskUI로 a와r를 입력한 경우 TaskCartManagement로 p를 입력한 경우 TaskPaymentProcessing으로 e를 입력한 경우엔 구현한 kernel function을 실행시키고 종료된다. 이때 command에 맞는 task를 resume하기 전 메시지를 큐에 저장한다.

다음은 void TaskPaymentProcessing(void* p_arg)에 대한 설명이다.

우선순위는 12이고 카트 내용의 대한 결제를 진행하기 위한 task이다. TaskInput에 의해 보내진 메시지를 확인한다. 이때 수신한 메시지에는 콘솔에서 입력된 결제 비밀번호가 함께 담겨있기 때문에 이를 이용해 결제를 처리한다. 이후 결제 비밀번호가 올바르고 결제 시점에서 카트에 담긴 물품이 구매가능하면 성공, 그렇지 않다면 결제를 실패한다.

다음은 void TaskCartManagement(void* p_arg)에 대한 설명이다.

우선순위는 13이고 카트 내역을 관리한다. TaskInput에서 보낸 메시지를 수신하고 메시지를 기반으로 카트에 넣을지 뺄지를 결정한다.

다음은 void TaskUI(void* p_arg)에 대한 설명이다.

우선순위는 14로 가장 낮고 전체 물품 리스트 혹은 카트 리스트를 TaskInput에서 보낸 메시지를 기반으로 출력한다.

-task간 semaphore와 message queue의 활용방안-

먼저 **semaphore**는 productList라는 구조체 배열에 접근하는 경우에 사용하였다. 전체적인 흐름을 봤을 때 TaskInput이 콘솔 입력에 따른 명령으로 task를 하나씩 실행시키므로 semaphore의 필요성이 없어 보일 수 있으나 우선순위가 A>B라고 할 때 B가 TaskInput에 의해 Resume되었고 critical section을 실행중일 때 우선순위가 가장 높은 TaskInput이 cpu 주도권을 가져가고 A를 실행시킨다면 B와 A가 race condition에 빠질 수 있다. 따라서 semaphore를 활용하여 이와 같은 상황을 해결하였다.

Message queue는 TaskInput에서 Task를 실행시킬 때 필요한 추가정보를 담기위해 사용하였다. 코드에서는 Message라는 구조체에 정보를 담고 메시지 큐에 저장한다. Message 구조체 내부에는 어떤 동작인지를 저장하는 MsgType, 그리고 카트를 관리할 때 필요한 정보를 담는 cartItem, 결제에 필요한 비밀번호를 담는 password 변수가 정의되어 있다.
또한 MsgType은 두 가지 명령을 처리하는 하나의 task에서 분기를 나누는 기준으로 사용된다.

-제안서 대비 변경사항-

먼저 다중 사용자에서 단일 사용자일때로 변경하였다.

다중 사용자에서 작동하도록 구현시에는 콘솔을 여러 개 띄워 같은 프로그램을 동작시키고 실험하려 했으나 이때 파일 동시접근에 대한 제어를 위해 윈도우 파일 시스템에 대한 세마포어를 사용해야 했기에 수정하였다.
따라서 파일 수정이 아닌 내부 task간 변수 동시접근에 대한 제어를 semaphore를 이용해 처리하였다.

다음은 kernel function에 대한 변경사항이다. 제안서에서는 OSTimeDlyHMSM을 수정하여 필요할 때 함수를 호출하는 방식으로 진행하려 했으나 구현해보는 과정에서 OSTaskSuspend와 OSTaskResume 그리고 우선순위를 적절히 활용하니 흐름에 문제가 발생하지 않았다. 따라서 OS_TASK.C 파일 내부에 command 실행 내역을 프로그램 종료시 출력하는 함수를 구현하였다. 구현한 코드에서 OSTaskResume은 Task switch를 담당하므로 kernel function은 OSTaskResume의 역할을 하면서 입력인자로 받은 세부내용과 Task 정보를 함께 저장한 후 종료시에 모두 출력하도록 하였다.

마지막으로 task에 대한 변경점이다.

TaskLoadProducts task는 반복 실행이 아닌 초기화 task로 한번만 실행되고 종료되게끔 변경되었다.

TaskInventoryManagement는 단일 사용자로 변경되면서 삭제되었다.

우선순위는 다음과 같이 변경되었다.

void TaskInput(void* p_arg); **우선순위 10 -> 11**

void TaskUI(void* p_arg); **우선순위 11 -> 14**

void TaskLoadProducts(void* p_arg); **우선순위 12 -> 10**

void TaskCartManagement(void* p_arg); **우선순위 13 -> 13**

void TaskPaymentProcessing(void* p_arg); **우선순위 14 -> 12**

프로그램의 흐름에 맞게 동작하기 위해 초기화 task로 변경된 TaskLoadProducts는 가장 높은 우선순위 10으로 변경되었다.
결제 프로세스를 구현하면서 실제로는 확인하기 어려우나 동시에 처리된다고 가정할때 결제 후 물품내용을 수정하는 것이 카트에 담기 전 물품내용을 확인하는 것보다 먼저 진행되어야 재고가 없는 물품을 카트에 담는 행위를 막을 수 있다고 생각하여 결제 task 우선순위를 상승시켰다. UI는 따라서 가장 낮은 우선순위로 바뀌었다.

-동작 예시-

```
C:\SOFTWARE\PROJECT>test.exe
Product list is initialized.
Loaded products:
Product Name: Milk, Price: 1500, Stock: 50, Location: Dairy
Product Name: Egg, Price: 2500, Stock: 40, Location: Dairy
Product Name: Bread, Price: 1500, Stock: 30, Location: Bakery
Product Name: Apple, Price: 1000, Stock: 100, Location: Fruit
Product Name: Chicken, Price: 5000, Stock: 20, Location: Meat
<Command List>
l: print list
a: add to cart
r: remove from cart
p: purchase cart item
```

초기 실행화면이다. 전체 상품 목록이 products.txt파일을 기반으로 초기화되고 콘솔에 출력된다. 이후 사용할 수 있는 command list가 표시된다.

```
input : a
Enter product name and quantity(ex: Apple 3)
Milk 30
input : a
Enter product name and quantity(ex: Apple 3)
Chicken 5
input : c
Current cart list:
Product Name: Milk, Count: 30, Price: 45000
Product Name: Chicken, Count: 5, Price: 25000
```

이후 a를 입력하여 각각 Milk 30개, Chicken 5개를 카트에 담는다. 그후 c를 이용하여 카트를 확인해보면 담긴 것을 확인할 수 있다.

```
input : r
Enter product name and quantity(ex: Apple 3)
Milk 10
input : c
Current cart list:
Product Name: Milk, Count: 20, Price: 30000
Product Name: Chicken, Count: 5, Price: 25000
```

이번엔 r를 입력하여 Milk 10개를 뺐다. 카트내역을 확인해보면 빠진 것을 확인한다.

```
input : a
Enter product name and quantity(ex: Apple 3)
Bread 40
Out of stock.
input : a
Enter product name and quantity(ex: Apple 3)
Banana 20
Product not found.
```

담기 요청한 수량이 너무 많거나 요청한 물품이 없다면 오류를 출력한다.

```
input : r
Enter product name and quantity(ex: Apple 3)
Chicken 10
There are fewer Chicken than your request.
input : r
Enter product name and quantity(ex: Apple 3)
banana 3
No banana found..
```

빼기 요청한 수량이 너무 많거나 요청한 물품이 없다면 마찬가지로 오류를 출력한다.

```
input : c
Current cart list:
Product Name: Milk, Count: 20, Price: 30000
Product Name: Chicken, Count: 5, Price: 25000
input : p
Please enter your 4-digit password:
0000
Passwords do not match..
input : p
Please enter your 4-digit password:
1234
Payment Successful.
```

담긴 카트 내용을 기반으로 결제를 진행하는 모습이다. 비밀번호가 일치하지 않는다면 오류를 반환하고 일치하면 결제되는 것을 확인한다.

```
input : c
Current cart list:
input : l
Current product list:
Product Name: Milk, Price: 1500, Stock: 30, Location: Dairy
Product Name: Egg, Price: 2500, Stock: 40, Location: Dairy
Product Name: Bread, Price: 1500, Stock: 30, Location: Bakery
Product Name: Apple, Price: 1000, Stock: 100, Location: Fruit
Product Name: Chicken, Price: 5000, Stock: 15, Location: Meat
```

이후 카트 내용은 초기화되고 l을 이용해 확인해보면 재고가 빠진 것을 확인할 수 있다.

```
input : a
Enter product name and quantity(ex: Apple 3)
Egg 30
input : c
Current cart list:
Product Name: Egg, Count: 30, Price: 75000
input : l
Current product list:
Product Name: Milk, Price: 1500, Stock: 30, Location: Dairy
Product Name: Egg, Price: 2500, Stock: 40, Location: Dairy
Product Name: Bread, Price: 1500, Stock: 30, Location: Bakery
Product Name: Apple, Price: 1000, Stock: 100, Location: Fruit
Product Name: Chicken, Price: 5000, Stock: 15, Location: Meat
input : a
Enter product name and quantity(ex: Apple 3)
Egg 30
input : c
Current cart list:
Product Name: Egg, Count: 60, Price: 150000
```

위는 예외처리 결과이다. 코드 구성상 카트에 담는 것은 재고에서 사라지지 않고 결제가 되어야 재고에서 빠진다. 따라서 전체 재고 수보다 낮은 값으로 여러 번 카트에 담게되면 전체 재고 수보다 더 많이 담을 수 있게된다.

```
input : p
Please enter your 4-digit password:
1234
Egg Out of stock
Payment Failure.
input : c
Current cart list:
Product Name: Egg, Count: 0, Price: 0
```

따라서 결제시 위와 같이 전체 재고보다 많은 수가 카트에 담겼고 이를 결제하려 하면 재고가 부족하다는 오류를 출력한다.

```
input : e
Task Execution History:
Task Priority: 13, Request: CART Milk 30 + Success
Task Priority: 13, Request: CART Chicken 5 + Success
Task Priority: 14, Request: UI
Task Priority: 13, Request: CART Milk 10 - Success
Task Priority: 14, Request: UI
Task Priority: 13, Request: CART Chicken 10 - Failure
Task Priority: 13, Request: CART banana 3 - Failure
Task Priority: 14, Request: UI
Task Priority: 12, Request: PAY   Failure
Task Priority: 12, Request: PAY   Success
Task Priority: 14, Request: UI
Task Priority: 14, Request: UI
Task Priority: 13, Request: CART Egg 30 + Success
Task Priority: 14, Request: UI
Task Priority: 14, Request: UI
Task Priority: 13, Request: CART Egg 30 + Success
Task Priority: 14, Request: UI
Task Priority: 14, Request: UI
Task Priority: 12, Request: PAY   Failure
Task Priority: 14, Request: UI
```

마지막으로 e를 누르게되면 프로그램이 종료되고 전체 실행내역을 kernel fucntion에 의해 출력하게 된다.