# Trenton Rochelle

9561606

# Homework #3

## CMPSC 170 (divyagrawal)

Winter 2019

Due Date: Friday February 22, 2019 (in HW Box by 4:30PM)

Q1. There are N processes in the system each of which may update a file. However, the updates to the file must be done in a critical section, i.e., only one process should be updating the file at any time. A simple solution is to require each process to execute the following code:

P(mutex);

Update(FILE);

V(mutex);

The drawback with the above solution is that the “wait queue” is managed implicitly by the system, and we do not know the policy used by the system to permit the waiting processes to enter the critical section. We want to use **First-come-First-served** policy. In order to do that, you have to modify the above solution so that the waiting queue is managed explicitly by your <entry-code> and <exit-code> for the critical section. For example, when the process tries to enter critical section it checks if the queue is empty. If yes it manipulates the queue and enters the critical section. Otherwise, it goes to sleep using semaphores. On exit from the critical section, the process does similar checks and wakes up an appropriate process. Certain CAVEATS:

* Use only Binary Semaphores.
* Clearly Indicate the initial values of all the semaphores in your solution.
* Also, the initial values of any shared variables and initial states of any shared data structures.

Develop a semaphore solution to the above synchronization problem.

s.value = 1;

P(s){

if(s.value==1){

s.value = 0;

}

else{

s.queue.push\_back(process)

process.block()

}

};

Update(FILE);

V(s){

if(s.queue.empty()){

s.value = 1;

}

else{

process = s.queue.front();

process.wakeup();

}

};

Q2. A barber-shop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served the barber goes to sleep. If a customer enters the barber-shop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, then the customer sits in one of the available free chairs. If the barber is asleep, the customer wakes the barber up. Write a monitor for solving the sleepy barber problem.

process CUSTOMER[1 to N]

{

while (true) {

let\_your\_hair\_grow();

call Barber\_Shop.get\_haircut();

}

}

process BARBER

{

while (true) {

call Barber\_Shop.get\_next\_customer();

haircut();

call Barber\_Shop.finished\_cut();

}

}

monitor Barber\_Shop

{

int barber = 0; # chair and open variables are not needed

cond barber\_available;

cond chair\_occupied;

cond door\_open;

cond customer\_left;

procedure get\_haircut()

{

while (barber==0) wait(barber\_available);

barber = barber – 1;

signal(chair\_occupied);

wait(door\_open);

signal(customer\_left);

}

procedure get\_next\_customer()

{

barber = barber + 1; signal(barber\_available);

wait(chair\_occupied);

}

procedure finished\_cut()

{

signal(door\_open);

wait(customer\_left);

}

}

Q3. Develop an implementation of monitors using fair semaphores. Specifically, show the pieces of code necessary for:

1. monitor entry
2. monitor exit
3. signal a condition variable
4. wait on a condition variable

Assume that the signaling discipline is ***signal-urgent***, i.e., a waiting process is woken up immediately. You may make the simplifying assumption that there is only one condition variable used per monitor.

The two interpretations of a signal operations are (assuming Q has been waiting for condition x and P performs signal.x):

1. P waits until Q either leaves the monitor, or waits for another condition;
2. Q waits until P either leaves the monitor, or waits for another condition;

1 is known as the ***signal-urgent*** mechanism while 2 is called ***signal-continue***.

sem e=1, unlocked=0, wait[]={0}  
int nw=0, waitlist[]={0}  
bool mLock = true  
  
entry(monitor):  
    P(e)  
    if mLock:  
        nw++  
        V(e)  
        P(unlocked)  
        nw--  
    mLock = true  
  
  
exit(monitor):  
    mLock = false  
    if (nw > 0): V(unlocked)  
    else: V(e)  
  
wait(monitor, cv):  
    V(e)  
    waitlist[cv] += 1  
    P(wait[cv])  
    waitlist[cv] -= 1  
    P(e)  
  
signal(monitor, cv):  
    if waitlist[cv] > 0: V(wait[cv])

Q4. Consider the following oil depot operation. A storage tank of capacity DepotCap acts as a transfer point for crude oil moving from tankers to trucks. Tankers come in different sizes and capacities but all trucks have the same capacity TruckCap (TruckCap < DepotCap). Furthermore, tankers can unload their contents piecemeal but trucks must be loaded to capacity in a single operation. The activities of tankers and trucks are represented by the following two prototype processes:

PROCESS cargo-ship-tanker;

LOOP

Contents 🡨 go\_to\_Oil\_Well\_and\_fill\_up();

Oil.Deposit(contents);

END LOOP;

END PROCESS cargo-ship-tanker;

PROCESS oil-truck;

LOOP

Oil.Retrieve();

go\_to\_gas\_stations\_and\_unload();

END LOOP;

END PROCESS oil\_truck;

Exhibit an implementation of the monitor “Oil” with external entry points “Deposit()” and “Retrieve()” which represent the operations of the storage tank. Assume that the tank is initially empty. Be sure to use mnemonic names for the variables and conditions. Remember that there are an arbitrary number of tankers and trucks. Your solution should avoid unnecessary waits for trucks or tankers.

Monitor OIL{

Total = 0

Full

Empty = True

TruckCap

DepotCap

Deposit(contents){

if (Total == DepotCap){

wait(Full)

}

tank.push(contents)

Total+=contents

if Total >= TruckCap{

notify(Empty)

}

}

Retrieve(){

if tank <= TruckCap{

Wait(Empty)

}

Total-=TruckCap

Amount = TruckCap

if Total == DepotCap

notify(Full)

}

Return Amount

}

Q5. The world has survived a terrible war, and civilization is in a collapse. Deep in the Australian dessert the small settlement called BARTERTOWN keeps the traditions of the good old days alive, by helping the survivors of the cataclysm trade goods for currency (silver coins). BARTERTOWN makes its living by taking a 10 percent commission from every deal.

If a client and a dealer want to make a deal, they first agree on a price. For example, ten dollars for a jug of water. Each dealer has a collection of “tickets”, and he tears one in half, keeping one half and giving the other half to the client. The dealer and client then independently go to the cashier. Assume that they arrive more or less at the same time, but in some random order. The dealer gives the cashier his half of the ticket and the product. The client gives the cashier the other half of the ticket and the cash, and then waits. The cashier’s job is to match up the client and the dealer (by matching their tickets together) and then carry out the transaction, as follows:

* If the price equals the cash provided by the client, then the cashier deducts 10% from the cash and throws it in a POT. The cashier gives the remaining 90% of the cash to the dealer and gives the product to the client.
* If the price is different from the cash provided by the client, then the cashier throws all the money in the POT, discards the product, and returns nothing to the client and the dealer. (This discourages cheating.)

Assume that clients and dealers are represented by processes and that BARTERTOWN is represented by a monitor. Given that at most N clients and N dealers are active at one time, develop a monitor with two entries, BARTER.BUY(ticket, amount) and BARTER.SELL(ticket, product, price), that implements the rules of trade given above. Assume that there are a small number of cashier processes that start by executing BARTER.CASHIER(), which never returns. Two tickets match if their values are the same. The price and amount are of type DOLLAR (the constant ZERO$ represents 0 dollars), and the product is of type STUFF (NOTHING represents nothing, for the error case).

Process Cashier(){

while(true){

BARTER.nextCustomer()

BARTER.process()

BARTER.endTransaction()

}

}

Monitor BARTER{

DEALER{

TICKET ticket

STUFF product

DOLLAR amount

DEALER DEALER(ticket, product, price){

DEALER newDealer

newDealer.ticket = ticket

newDealer.product = product

newDealer.amount = price

return newDealer

}

}

BUYER{

TICKET ticket

DOLLAR amount

BUYER BUYER(ticket,amount){

BUYER newBuyer

newBuyer.ticket = ticket

newBuyer.ammount = amount

return newBuyer

}

}

TRANSACTION{

TICKET ticket

DOLLAR amount\_seller

DOLLAR amount\_buyer

STUFF product

boolean seller\_in = False

boolean buyer\_in = False

matchedTicket = False

TRANSACTION TRANSACTION(ticket,stuff,amount){

TRANSACTION newTransaction

newTransaction.ticket = ticket

newTransaction.stuff = stuff

newTransaction.amount\_seller = amount

boolean seller\_in = True

}

TRANSACTION TRANSACTION(ticket,amount){

newTransaction.ticket = ticket

newTransaction.amount\_buyer = amount

boolean buyer\_in = True

}

}

TRANS\_MAP{

map<TRANSACTION> transactions[large\_prime]

void insertSeller(ticket,product,amount){

int i = getHash(ticket)

transactions[i] = TRANSACTION(ticket,product,amount)

}

void insertBuyer(ticket,amount){

int i = getHash(ticket)

transactions[i] = TRANSACTION(ticket.amount)

}

boolean getIndex(ticket){

int i = getHash(ticket)

if(transactions[i]==NULL)

return -1

else

return i

}

boolean both\_in(ticket){

int i = getHash(ticket)

return (transactions[i].buyer\_in && transactions[i].seller\_in)

}

DOLLAR getMoney(ticket){

i = getIndex(ticket)

DOLLAR money = transactions[i].amount

transactions[i].amount = $ZERO

if(transactions[i].product==NULL){

transactions[i] = NULL

}

transactions[i].buyer\_in = False

return money

}

STUFF getProduct(ticket){

i = getIndex(ticket)

STUFF product = transactions[i].stuff

transactions[i].stuff = NULL

if(transactions[i].amount==NULL){

transactions[i] = NULL

}

transactions[i].seller\_in = False

return product

}

}

queue<ticket> tickets

queue<int> tickets\_index

int cashiers

TRANS\_MAP transactions

DOLLAR POT

ticket newBuyer(ticket,amount){

transactions.insertBuyer(ticket,amount)

return transactions.getIndex(ticket)

}

ticket newSeller(ticket,amount){

transactions.insertSeller(ticket,product,price)

return transactions.getIndex(ticket)

}

boolean checkTicket(ticket){

return transactions.checkTicket(ticket)

}

STUFF getProduct(ticket){

return transactions.getProduct(ticket)

}

DOLLAR getMoney(ticket){

return transactions.getMoney(ticket)

}

STUFF BUY(ticket,amount){

while(cashiers==0) wait(cashier\_available);

tickets.push\_front(newBuyer(ticket,amount))

cashiers = cashiers - 1

notify(newCustomer)

while(checkTicket(ticket)){

wait(matchedTicket)

}

STUFF product = getProduct(ticket)

notify(completeTransaction)

return product

}

DOLLAR SELL(ticket,product,price){

while(cashiers==0) wait(cashier\_available);

tickets.push\_front(newSeller(ticket,product,price))

cashiers = cashiers - 1

notify(newCustomer)

while(!checkTicket(ticket)){

wait(matchedTicket)

}

DOLLAR money = getMoney(ticket)

notify(completeTransaction)

return money

}

nextCustomer(){

cashier = cashier + 1

notify(casher\_available)

wait(newCustomer)

}

process(){

for(int i=0,i<len(tickets),i++)){

ticket = tickets[i]

if(transactions.both\_in(ticket)){

int i = transactions.getIndex(ticket)

if(transactions[i].amount\_buyer==transactions[i].amount\_seller){

POT = POT + .1 \* transactions[i].amount\_seller

transactions[i].amount\_seller = .9 \* transactions[i].amount\_seller

transactions[i].matchedTicket = True

tickets\_index.push\_front(i)

continue

}

else{

POT = POT +transactions[i].amount\_seller

transactions[i].amount\_seller = $ZERO

transactions[i].product = NOTHING

transactions[i].matchedTicket = True

tickets\_index.push\_front(i)

continue

}

}

}

}

endTransaction(){

notify(matchedTicket)

wait(completeTransaction)

int i = tickets\_index.front()

tickets\_index.pop()

tickets[i].remove()

}

}