

Saiful Islam

Task5 - CSC33200(L)-Class #34293-Islam

### **Lab 5 Report (Critical section of the code and logical errors)**

#### **Prevent race conditions**

**Prevent a process from continuously requesting access to the shared memory.**

**Prevent unnecessary cpu cycle.**

With my solution, the `balance.txt` file is being read and written by no more than a single user at any given time. That being said, the parent, first child, or second child can read the file or write into the file if and only if none of the other processes are currently reading or writing into the file.

For example, a son should not be able to withdraw while the parent is depositing, as the new balance will become inconsistent with what the balance is actually supposed to be. Instead, in this situation, the son should request and wait.

Similarly, while one son is viewing and withdrawing from the balance, the other son should not be able to withdraw and the parent should not be able to deposit money as these things also lead to inconsistent balance. Instead, they should request and wait.

Another case of preventing race conditions is in maintaining the number of attempts so a son should wait while the other is withdrawing (only one son should go at a time).

Also, to prevent unnecessary CPU cycles, the process requesting to view the balance is blocked until the process currently working with the balance is done with what they are doing. The process does not continuously request access but requests access only once and is given the ATM when their time comes.

To solve these problems, two semaphores are used.

```

1
2 // either son or parent any given time so that balances match up
3 sem_create(parentOrSon, 1);
4
5 // only one son can go at a time
6 // 0 means a son is currently at the ATM, 1 means no son currently at the ATM
7 sem_create(oneSonAtATime, 1);

```

Critical section (and logical errors) and the fix:

```

if (pid == CHILD)
{
    //First Child Process. Dad tries to do some updates.
    printf("Dad's Pid: %d\n", getpid());
    N = NumOfDepositAttempt;
    for (i = 1; i <= N; i++)
    {
        printf("Dad is requesting to view the balance.\n"); //Dad is requesting to get hold of an ATM.
        P(parentOrSon);
        fp1 = fopen("balance.txt", "r+"); //Dad successfully got hold of the ATM.
        fscanf(fp1, "%d", &bal2);
        printf("Dad reads balance = %d \n", bal2);
        int r = rand() % 5 + 1;
        printf("Dad needs %d sec to prepare money\n", r);
        sleep(r); //Dad Process is sleeping for r sec. You need to make sure that other processes can work in the mean time.

        fseek(fp1, 0L, 0); //Dad will now deposit the money. For this Dad will access the ATM again. And update the current balance.
        bal2 += DepositAmount;
        fprintf(fp1, "%d \n", bal2);
        fclose(fp1);
        for (size_t d = 0; d < DepositAmount; d++)
        {
            V(balance);
        }
        printf("Dad writes new balance = %d \n", bal2);
        printf("Dad will deposit %d more time\n", N - i); //Dad deposited the money.
        V(parentOrSon);
        sleep(rand() % 10 + 1); /* Dad will wait some time for requesting to see balance again.*/
    }
}

```

```

// Second child process: first part son tries to do withdraw
flag = FALSE;
while (flag == FALSE)
{
    P(oneSonAtATime); // block other son from performing withdrawals or wait for other son to finish performing withdrawal
    printf("SON_1 is requesting to view the balance.\n"); //Son_1 is requesting to get hold of the ATM.
    fp3 = fopen("attempt.txt", "r+"); //Son_1 successfully got hold of the ATM.
    fscanf(fp3, "%d", &N_Att); // Son_1 Checks if he has more than 0 attempt remaining.
    printf("Attempt remaining: %d.\n", N_Att);
    if (N_Att == 0)
    {
        fclose(fp3);
        flag = TRUE;
    }
    else
    {
        for (size_t w = 0; w < WithdrawAmount; w++)
        {
            P(balance);
            P(parentOrSon); // block here so parent does not deposit in the middle of withdrawal and child does not withdraw in the middle of deposit
            fp2 = fopen("balance.txt", "r+"); //Son_1 reads the balance.
            fscanf(fp2, "%d", &bal2);
            printf("SON_1 reads balance. Available Balance: %d \n", bal2);
            printf("SON_1 wants to withdraw money. "); //And if balance is greater than Withdraw amount, then son can withdraw money.
            fseek(fp2, 0L, 0);
            bal2 -= WithdrawAmount;
            fprintf(fp2, "%d\n", bal2);
            fclose(fp2);
            V(parentOrSon);
            printf("SON_1 withdrew %d. New Balance: %d \n", WithdrawAmount, bal2);

            fseek(fp3, 0L, 0); //SON_1 will write the number of attempt remaining in the attempt.txt file.
            N_Att -= 1;
            fprintf(fp3, "%d\n", N_Att);
            fclose(fp3);
            printf("Number of attempts remaining: %d \n", N_Att);
        }
    }
    V(oneSonAtATime); // allow other son to perform withdrawal
    sleep(rand() % 10 + 1); //SON_1 will wait some time before the next request.
}

while (flag1 == FALSE)
{
    P(oneSonAtATime); // block other son from performing withdrawals or wait for other son to finish performing withdrawal
    printf("SON_2 is requesting to view the balance.\n"); //Son_2 is requesting to get hold of the ATM.
    fp3 = fopen("attempt.txt", "r+"); //Son_2 successfully got hold of the ATM.
    fscanf(fp3, "%d", &N_Att); // Son_2 Checks if he has more than 0 attempt remaining.
    printf("Attempt remaining: %d.\n", N_Att);
    if (N_Att == 0)
    {
        fclose(fp3);
        flag1 = TRUE;
    }
    else
    {
        for (size_t w = 0; w < WithdrawAmount; w++)
        {
            P(balance);
            P(parentOrSon); // block here so parent does not deposit in the middle of withdrawal and child does not withdraw in the middle of deposit
            fp2 = fopen("balance.txt", "r+"); //Son_2 reads the balance.
            fscanf(fp2, "%d", &bal2);
            printf("SON_2 reads balance. Available Balance: %d \n", bal2);
            printf("SON_2 wants to withdraw money. "); //And if balance is greater than Withdraw amount, then son can withdraw money.
            fseek(fp2, 0L, 0);
            bal2 -= WithdrawAmount;
            fprintf(fp2, "%d\n", bal2);
            fclose(fp2);
            V(parentOrSon);
            printf("SON_2 withdrew %d. New Balance: %d \n", WithdrawAmount, bal2);

            fseek(fp3, 0L, 0); //SON_2 will write the number of attempt remaining in the attempt.txt file.
            N_Att -= 1;
            fprintf(fp3, "%d\n", N_Att);
            fclose(fp3);
            printf("Number of attempts remaining: %d \n", N_Att);
        }
    }
    V(oneSonAtATime); // allow other son to perform withdrawal
    sleep(rand() % 10 + 1); //SON_2 will wait some time before the next request.
}
}

```

The control is given to other processes using the v operation.

**Prevent a son from withdrawing money when there is no balance**

**Prevent undefined outputs like negative balance**

To solve the problem of having a negative balance, the son that is requesting to view the balance must wait until the current balance is greater than or equal to the son's request; the son should

not continuously check if the balance is available to withdraw from. The sons should wait until enough balance is available and only then they will withdraw money.

To solve the problem, one semaphore is used where the current value of the semaphore represents the current balance.



```
1 // don't allow for negative balances
2 sem_create(balance, initBalance);
```

This semaphore is incremented by the deposit amount (in the parent) and decremented by the withdrawal amount (in both children), resulting in synchronization between the processes so that a son does not withdraw when there is not enough balance.

Critical section (and logical errors) and the fix:



```
1 for (size_t d = 0; d < DepositAmount; d++)
2     V(balance);
```

```

fseek(fp1, 0L, 0); //Dad will now deposit the money. For this Dad will access the ATM again
bal2 += DepositAmount;
fprintf(fp1, "%d \n", bal2);
fclose(fp1);
for (size_t d = 0; d < DepositAmount; d++)
    V(balance);
printf("Dad writes new balance = %d \n", bal2);
printf("Dad will deposit %d more time\n", N - i); //Dad deposited the money.
V(parentOrSon);
sleep(rand() % 10 + 1); /* Dad will wait some time for requesting to see balance again.*/

```



```

1  for (size_t w = 0; w < WithdrawAmount; w++)
2      P(balance);

```

```

else
{
    for (size_t w = 0; w < WithdrawAmount; w++)
        P(balance);
    P(parentOrSon); // block here so parent does not
    fp2 = fopen("balance.txt", "r+"); //Son_1 reads the balance.
    fscanf(fp2, "%d", &bal2);
    printf("SON_1 reads balance. Available Balance: %d \n", bal2);
    printf("SON_1 wants to withdraw money. "); //And if balance is greater than withdraw amount
    fseek(fp2, 0L, 0);
}

```

```


else
{
    for (size_t w = 0; w < WithdrawAmount; w++)
        P(balance);
    P(parentOrSon); // block here so parent does not
    fp2 = fopen("balance.txt", "r+"); //Son_2 reads the balance.
    fscanf(fp2, "%d", &bal2);
    printf("SON_2 reads balance. Available Balance: %d \n", bal2);
    printf("SON_2 wants to withdraw money. "); //And if balance is greater than withdraw amount
    fseek(fp2, 0L, 0);
}

```

**Program does not terminate**


From the way the initial amounts are provided, the program does not terminate as the sons will still be requesting money and have \$100 left to request (as they will request for a total of \$400). However, the parent will only deposit a total of \$300, not satisfying the sons. To solve this problem, I changed the withdrawal amount for the sons. Alternatively, the initial balance could have been set to \$100 instead of \$0.

Critical section (and logical errors) before the fix:



```
1  #define DepositAmount    60      /* The amount of money Dad deposits at a time */
2  #define WithdrawAmount  20      /* The amount of money Son withdraws at a time */
```

Critical section (and logical errors) after the fix:



```
1  #define DepositAmount 60 /* The amount of money Dad deposits at a time */
2  #define WithdrawAmount 15 /* The amount of money Son withdraws at a time */
```

### Solved with as few semaphore variables as possible

I used three semaphore variables to solve these problems as they were all necessary. The first semaphore `balance` is used to prevent negative balances. The second semaphore `parentOrSon` is used to make sure that while the parent is depositing, a child cannot withdraw, and while a child is withdrawing, the parent cannot withdraw (prevents race conditions that result in inconsistent balance). The third and final semaphore `oneSonAtATime` is used to make sure that there are two ATM machines being used and prevent race conditions in changing the number of attempts. The semaphores `parentOrSon` and `oneSonAtATime` cannot be combined into one otherwise both sons can change the number of attempts at the same time or the number of ATM machines used will be at most one or three depending on how the one semaphore is utilized. We want to take advantage of all machines (so not use only one machine) and don't want to use more ATM machines than we have (so don't use three machines). Also, if the two semaphores are combined into one, both children can be on the two ATMs, and if there is insufficient balance, the parent

cannot deposit since both ATMs are taken up (another problem with having only two semaphores).