* + 1. **Höhere Technische Bundeslehranstalt Hollabrunn**
  1. **Höhere Abteilung für Elektronik – Technische Informatik**

|  |  |
| --- | --- |
| Klasse/ Jahrgang:  5BHEL | Übungsleiter:  REJ STO |
| Übungsnummer:  P2 | Übungstitel:  Producer Consumer Problem |
| Datum der Abgabe:  22.12.2016 | Teilnehmer:  Rockenbauer, Trucka |
| Unterschrift Schüler: | Unterschrift Lehrer: |

Inhaltsverzeichnis

[1.)Angabe 3](#_Toc470180999)

[1.1)Allgemeines 4](#_Toc470181000)

[2.)Funktionsbeschreibung 4](#_Toc470181001)

[2.1)Allgemein 4](#_Toc470181002)

[2.2) Hinzufügen eines Items 5](#_Toc470181003)

[2.3) Entfernen eines Items 5](#_Toc470181004)

[3.)Lösung des Synchronisationsproblems 6](#_Toc470181005)

[3.1)Lösung mit Mutex 6](#_Toc470181006)

[3.1.1)Code 6](#_Toc470181007)

[3.2)Lösung des Synchronisationsproblems mit Semaphore 9](#_Toc470181008)

[3.2.1)Code 9](#_Toc470181009)

[4.)Git Verwaltung: 13](#_Toc470181010)

[5.)Erkenntnis 15](#_Toc470181011)

Angabe einscannen

**Producer Consumer Problem**

# 1.)Angabe

Ziel dieses Projekts ist es, das Producer Consumer Problem mit Threads in der Programmiersprache C in Linux zu lösen. Zum einen sollte das Problem mit Mutex und Bedingungsvariable und zum anderen mit Semaphore gelöst werden.

Die Versionsverwaltung und Versionierung ist mit Git zu gestalten.

Pseudocode wie das Programm grundlegend funktionieren soll:

* Basic producer code:

*event = waitForEvent()*

*buffer.add(event)*

* Basic consumer code:

*event = buffer.get()*

*event.process()*

## 1.1)Allgemeines

Zuerst wurde in das C Programm eine Library eingebunden, um Threads erstellen zu können.



Um das Programm kompilieren zu können, muss ein eigener Aufruf für die Übersetzung in der Konsole angegeben werden.



# 2.)Funktionsbeschreibung

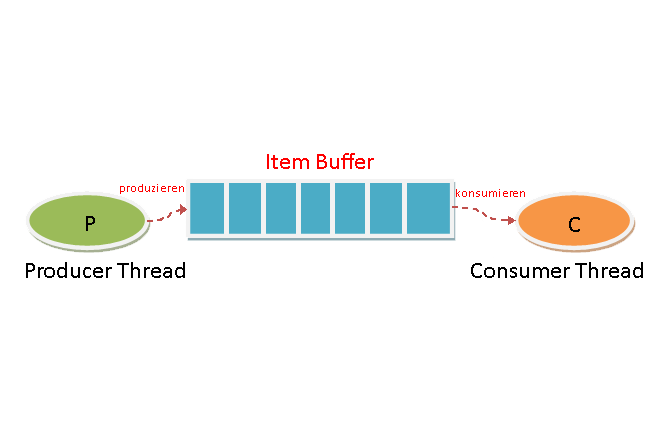
## 2.1)Allgemein

Das Programm besteht grundsätzlich aus zwei Teilen: Erstens dem **Producer**, welcher ein Item (Zufällig Generierte Zahl) **produziert** und es einem **Lager** (itemBuffer Array) hinzufügt.  
Und Zweitens aus dem **Consumer**, der ein Item aus dem **Lager** **konsumiert.**

Der **Producer** wartet immer auf ein Event, um ein Item zu produzieren. Als Event wurde der Input der Zahl 1 gewählt.  
Der **Consumer** wird alle 2 Sekunden ein Item aus dem Lager entfernen zu versuchen und somit ein Item konsumieren.

Befinden sich keine Items im Lager, so kann der **Consumer nichts konsumieren.**  
Befinden sich die maximale Anzahl an Items im Lager (Lager voll) so kann der **Producer nichts produzieren.**

Um nun das Problem zu lösen, wurden einerseits **Mutex** und andererseits **Semaphore** verwendet.

Die **Kritische Zone** in dem Programm ist das hinzufügen und entfernen von Items zu / aus dem Lager.

Kritische Zone

## 

## 2.2) Hinzufügen eines Items

Wird ein Item durch den Producer zum Lager hinzugefügt, so wird zuerst geprüft, ob sich im Lager Platz für weitere Items befindet.  
Ist genug Platz für ein weiteres, so wird dem itemBufferr (Array) das Item (Zufällig Generierte Zahl als Übergabeparameter des Unterprogramms) hinzugefügt und ein Counter (repräsentiert die Anzahl der Items im Lager) inkrementiert.

Ist nicht genügend Platz, so wird kein weiteres Item zum Lager hinzugefügt und eine Nachricht ausgegeben.

## 2.3) Entfernen eines Items

Wird ein Item durch den Consumer aus dem Lager entfernt, so wird zuerst geprüft, ob es im Lager mindestens 1 Item befindet.  
Existieren Items im Lager, so wird das oberste Item (Element) aus dem itemBuffer (Array) entfernt und ein Counter(repräsentiert die Anzahl der Items im Lager) dekrementiert.  
Existieren keine Items im Lager, so können keine Items konsumiert werden, und es wird eine Nachricht ausgegeben.

# 3.)Lösung des Synchronisationsproblems

## 3.1)Lösung mit Mutex

Beim Eintritt eines Threads in die Kritische Zone wird dieser Bereich gesperrt, mit **mutex\_lock**.   
Beim Verlassen eines Threads der Kritischen Zone wird der Bereich wieder freigegeben, mit **mutex\_unlock.**

Würden nun 2 (oder mehr) Threads gleichzeitig in die Kritische Zone eintreten wollen, so sperrt einer der Threads den Bereich zuerst und der/die anderen Threads warten, bis die Kritische Zone wieder freigegeben ist.

### 3.1.1)Code

#include<stdio.h>  
#include<string.h>  
#include<pthread.h>  
#include<stdlib.h>  
#include<unistd.h>  
  
pthread\_mutex\_t lock; // Mutex Variable  
int counter = 0; // Produktanzahl  
int lagerGrosse = 10; // Lagergröße  
int itemBuffer[10]; // Array für Lager  
//Prototypen  
void producer(void \*ptr);  
void consumer(void \*ptr);  
void insertItem(int item);  
void removeItem(int \*item);   
  
//Producer Unterprogramm  
void producer(void \*tid)   
{  
 while(1)   
 {  
 int event; //Variable für Eingabe  
 scanf("%d",&event); //Einlesen des Events  
 if(event == 1) //Ausführen wenn Event vorhanden  
 {  
 pthread\_mutex\_lock(&lock); //Mutex sperren  
 int item = rand()+1; //Random Wert (+1 damit nicht 0 vorkommt)  
 printf("\n Produktion gestartet\n");  
 insertItem(item); //Aufruf von insertItem  
 printf("\n Produktion beendet\n");  
 printf("\n produziertes item: #%d\n",item); //Produziertes Item anzeigen  
 printf("-------------------------\n");  
  
 pthread\_mutex\_unlock(&lock); //Mutex freigeben  
 }   
 }   
}  
  
//Consumer Unterprogramm   
void consumer(void \*tid)  
{  
 while(1)  
 {  
 sleep(2); //Alle 2 Sekunden consumer ausführen  
 pthread\_mutex\_lock(&lock); //Mutex sperren  
 int item;  
 printf("\n Konsum gestartet\n");  
 removeItem(&item); //Aufruf von removeItem  
 printf("\n Konsum beendet\n");  
 printf("\n konsumiertes item: #%d\n",item); //konsumiertes Item anzeigen  
 printf("-------------------------\n");   
  
 pthread\_mutex\_unlock(&lock); //Mutex freigeben  
 }   
}  
  
//insertItem Unterprogramm  
void insertItem(int item){  
 if (counter < lagerGrosse){ //Im Lager befindet sich Platz für weiter Items  
 itemBuffer[counter] = item; //Item in Lager(Array) platzieren  
 counter ++; //Anzahl der vorhandenen Items inkrementieren  
 printf("\n Item zum Lager hinzugefügt: %d items im Lager\n",counter); //Azahl der Items im Lager anzeigen  
 }else{  
 printf("\n Lager ist Voll\n");  
 }  
}  
  
//removeItem Unterprogramm  
void removeItem(int \*item){  
 if (counter > 0){ //Im Lager ist mindestens ein Item vorhanden  
 \*item = itemBuffer[counter-1]; //Item aus Lager(Array) entfernen  
 counter --; //Anzahl der vorhanden Items dekrementieren  
 printf("\n Item konsumiert: %d items im Lager\n", counter); //Anzahl der Items im Lager anzeigen  
 }else{  
 printf("\n Lager ist leer\n");  
 \*item = 0; //Itemwert auf 0 setzen   
 }  
}  
  
//Hauptprogramm  
int main(void)  
{  
 int id[2]; //Array für thread id's  
 pthread\_t thread\_1; //thread 1  
 pthread\_t thread\_2; //thread 2  
 int err; //err Variable für error handling   
  
 pthread\_mutex\_init(&lock, NULL);// Mutex initialisieren  
   
 //Producer thread erstellen  
 err = pthread\_create(&thread\_1, NULL, (void \*) &producer, (void \*) &id[0]);  
 if (err != 0)  
 printf("\ncan't create thread :[%s]", strerror(err));  
   
 //Consumer thread erstellen   
 err = pthread\_create(&thread\_2, NULL, (void \*) &consumer, (void \*) &id[1]);  
 if (err != 0)  
 printf("\ncan't create thread :[%s]", strerror(err));  
  
 // Wartet auf terminierung der threads  
 pthread\_join(thread\_1, NULL);  
 pthread\_join(thread\_2, NULL);  
  
 pthread\_mutex\_destroy(&lock); //Mutex auflösen  
 return 0;  
}

## 3.2)Lösung des Synchronisationsproblems mit Semaphore

Wenn ein Thread in die kritische Zone eintreten will darf der Zähler der Semaphore nicht 0 sein. Trifft das zu tritt der Thread in die kritische Zone ein, ansonsten wartet er bis der Zähler wieder erhöht wird. **sem\_wait**

Im Programm wurden zwei Semaphoren verwendet, eine für den producer und eine für den consumer. Wird ein item produziert wird der Zähler der Semaphore des consumers mit **sem\_post**  erhöht. Wird ein item konsumiert wird der Zähler der Semaphore des producers erhöht.

Somit ist es unmöglich dass ein Thread produziert wenn das Lager voll ist bzw. konsumiert wenn das Lager leer ist.

### 3.2.1)Code

#include<stdio.h>  
#include<string.h>  
#include<pthread.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<semaphore.h>  
  
sem\_t cEmpty; //semaphore für producer  
sem\_t cFull; //semaphore für consumer

pthread\_mutex\_t lock; //Mutex variable  
int counter = 0; // Produktanzahl  
int lagerGrosse = 10; //lagergröße  
int itemBuffer[10]; //Array für Lager  
  
  
//Prototypen  
void producer(void \*ptr);  
void consumer(void \*ptr);  
void insertItem(int item);  
void removeItem(int \*item);   
  
//unterprogramm producer  
void producer(void \*ptr)   
{  
 while(1)  
 {  
 int event; //variable für eingabe  
 scanf("%d",&event); //einlesen des events  
 if(event == 1) //ausführen wenn event vorhanden  
 {  
 sem\_wait(&cEmpty); //semaphore cEmpty sperren wenn nicht 0 (bei eintritt wir cEmpty decrementiert)

pthread\_mutex\_lock(&lock); //Mutex sperren  
 int item = rand()+1; //Random Wert (+1 damit nicht 0 vorkommt)  
 printf("\n Produktion gestartet\n");  
 insertItem(item); //insertItem aufruf  
 printf("\n Produktion beendet\n");  
 printf("\n produziertes item: #%d\n",item); //produziertes item anzeigen  
 printf("-------------------------\n");

pthread\_mutex\_unlock(&lock); //Mutex freigeben  
 sem\_post(&cFull); //cFull inkrementieren  
 }   
 }   
}  
  
//unterprogramm consumer   
void consumer(void \*ptr)  
{  
 while(1)  
 {  
 sleep(2); //alle 2 sekunden consumer ausführen  
 printf("\n Konsumer wartet auf Eintritt in kritischen Bereich\n");  
 sem\_wait(&cFull); //semaphore cFull sperren wenn nicht 0 (bei eintritt wir cFull decrementiert)

pthread\_mutex\_lock(&lock); //Mutex sperren  
 printf("\n Konsumer ist in kritischen Bereich eingetretten\n");  
 int item;  
 printf("\n Konsum gestartet\n");  
 removeItem(&item); //removeItem aufruf  
 printf("\n Konsum beendet\n");  
 printf("\n konsumiertes item: #%d\n",item); //konsumiertes item anzeigen  
 printf("-------------------------\n");

pthread\_mutex\_unlock(&lock); //Mutex freigeben  
 sem\_post(&cEmpty); //cEmpty inkrementieren  
 }   
}  
  
//unterprogramm insertItem  
void insertItem(int item){  
 if (counter < lagerGrosse){ //Im Lager befindet sich Platz für weiter Items  
 itemBuffer[counter] = item; //item im lager(array) platzieren  
 counter ++; //anzahl der vorhanden items inkrementieren  
 printf("\n Item zum Lager hinzugefügt: %d items im Lager\n",counter); //anzahl der items im lager anzeigen  
 }else{  
 printf("\n Lager ist Voll\n");  
 }  
}  
  
//unterprogramm removeItem  
void removeItem(int \*item){  
 if (counter > 0){ //im lager bedindet sich mindestens ein item  
 \*item = itemBuffer[counter-1]; //item aus lager(array) entfernen  
 counter --; //anzahl der vorhanden items dekrementieren  
 printf("\n Item konsumiert: %d items im Lager\n", counter); //anzahl der items im lager anzeigen  
 }else{  
 printf("\n Lager ist leer\n");  
 \*item = 0; //itemwert auf 0 setzen  
 }  
}  
  
//hauptprogramm  
int main(void)  
{  
 int i[2]; //array für thread id's  
 pthread\_t thread\_1; //thread 1  
 pthread\_t thread\_2; //thread 2  
 int err; //err Variable für error handling   
  
 sem\_init(&cFull, 0, 0); //semaphore cFull initialisieren  
 sem\_init(&cEmpty, 0, 10); //semaphore cEmpty initialisieren

pthread\_mutex\_init(&lock, NULL); //Mutex initialisieren

//producer thread erstellen  
 err = pthread\_create(&thread\_1, NULL, (void \*) &producer, (void \*) &i[0]);  
 if (err != 0)  
 printf("\ncan't create thread :[%s]", strerror(err));  
   
 //consumer thread erstellen   
 err = pthread\_create(&thread\_2, NULL, (void \*) &consumer, (void \*) &i[1]);  
 if (err != 0)  
 printf("\ncan't create thread :[%s]", strerror(err));  
  
 //wartet auf terminierung der threads  
 pthread\_join(thread\_1, NULL);  
 pthread\_join(thread\_2, NULL);  
  
 return 0;  
}

# 4.)Git Verwaltung:

Git dient zur Versionsverwaltung von Daten. Die Funktion beruht darauf dass es ein globales und ein lokales Repository gibt. Gearbeitet wird im lokalen. Kommt es zu Änderungen von Daten können diese auf das globale Repository hochgeladen werden und auch wieder heruntergeladen werden. Es kann auch mehrere lokale Repositories geben.

**Anlegen der Repositories:**

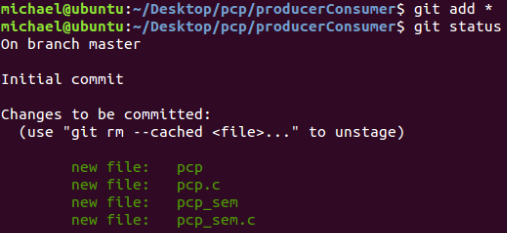
Zuerst wurde ein Git Repository erstellt.





Die beiden Programme wurden zu dem Repository hinzugefügt.

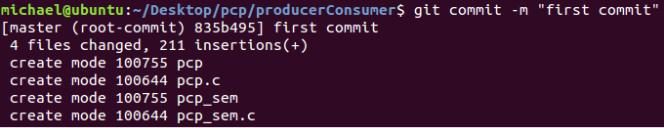




Dieser Befehl fügt die Dateien jedoch nur zum Index hinzu. Der Index ist eine Art Zwischenspeicher bevor die Änderungen endgültig bestätigt werden.

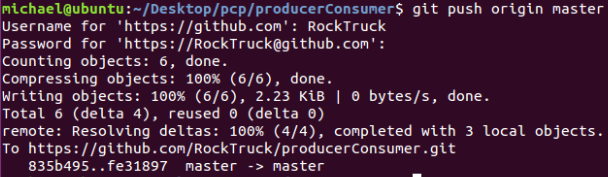
Bestätigt wird der Index mit commit.





Nun kann das lokale repository in das globale auf github hochgeladen werden





Anstelle des Masters kann auch ein Branch hochgeladen werden. Ein Branch ist sozusagen eine lokale Zwischenversion die bearbeitet wird und später wieder mit dem Master gemerged werden kann.

Dieser Befehl erstellt einen neuen Branch namens feature\_x



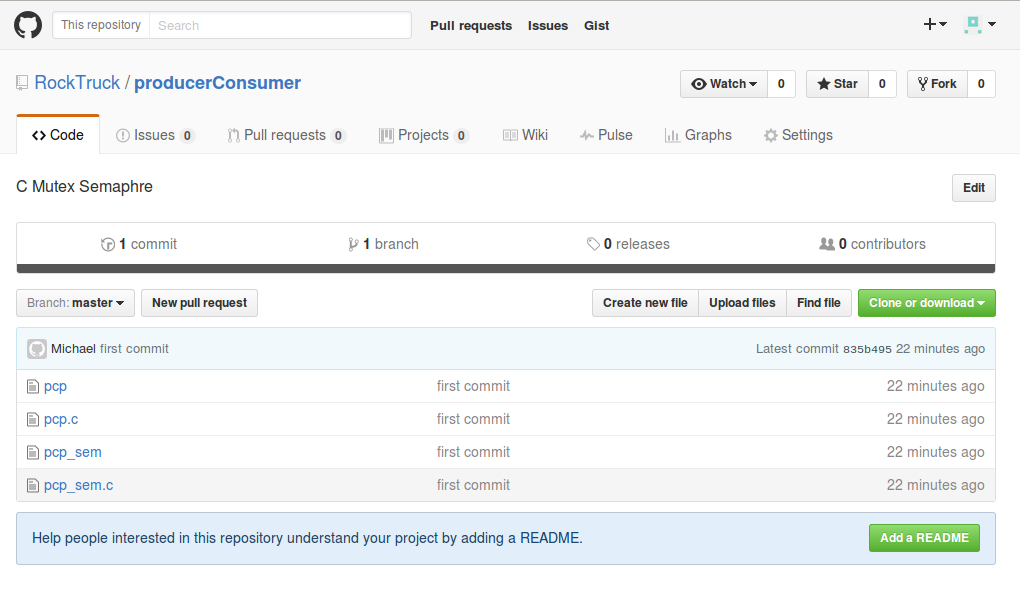
Dieser Befehl führt den Branch wieder mit dem Master zusammen



Um die aktuelle Version aus dem Repository zu erhalten wurde der Befehl pull verwendet.





**Repository auf GitHub**

# 5.)Erkenntnis

Um diese Aufgabe lösen zu können, wurde folgendes Wissen angeeignet.

* Programmieren von Threads
* Synchronisationsverfahren Mutex und Semaphore
* Git Versionsverwaltung