Datorteknik och Realtidssystem
Fö2

Shared Resources

LINKÖPINGS UNIVERSITET

Agenda

- Shared Resources
- C-programmering
- Lab 1
- Labmiljö : Terminalprogram, GitLab

Shared Resources

Shared Resources

En gemensam resurs är något som används av flera processer

Det kan t ex vara:

- Data : variabel, buffer, databas
- Hårdvara : display, timer, A/D-omvandlare, kärnan i en CPU

Critical Regions

En kritisk region är en sekvens i programkoden för en process där en gemensam resurs används.

Kritiska regioner kan vara:

- Relativt odelbara
  Processen får avbrytas, men ingen annan process får använda den gemensamma resursen
- Absolut odelbara
  Processen får INTE ens avbrytas.
  Avbrott stängs av under den kritiska regionen. Detta görs t ex i realtidssystemets kärnfunktioner.

Mutual Exclusion

Mutual exclusion (ömsesidig uteslutning) innebär att endast en process åt gången får använda den gemensamma resursen.

Förkortas som ”mutex”

Readers-Writer(s)

Då endast en process åt gången skriver till en gemensam resurs så kan flera andra processer läsa samma resurs utan ömsesidig uteslutning, under förutsättning att läsningen kan ske atomärt.

T ex nuvarande klockslag. En process uppdaterar klockan och flera andra processer kan läsa av klockvärdet.

Atomär avläsning innebär att hela datamängden kan läsas konsistent utan uppdelning.

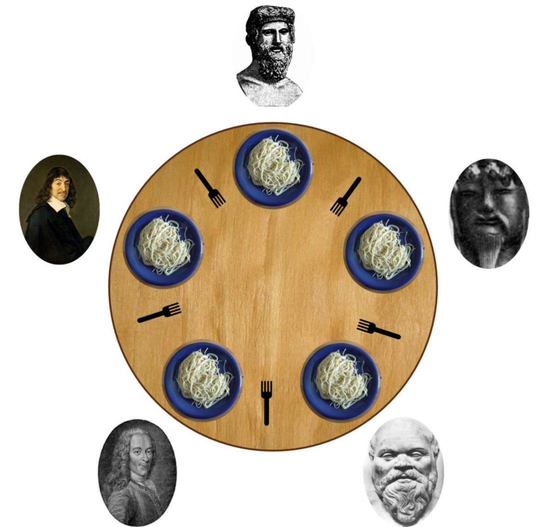
[09:59 -> 10:00 på 8-bitars processor]

```
09:59 : Avläsning 09
10:00 : Avbrott/Uppdatering
10:00 : Avläsning 00
```

Icke atomär avläsning,
Klockan läses som 09:00

Ätande filosofer

Problemet består i att konstruera ett beteende/algoritm så att alla filosofer kan äta alternativt tänka oberoende av varandra.



1. Tänk tills vänster gaffel är ledig, sen ta upp den
2. Tänk tills höger gaffel är ledig, sen ta upp den
3. Med båda gafflarna, ät under en bestämd tid
4. Lägg ned höger gaffel
5. Lägg ned vänster gaffel
6. Börja om från början

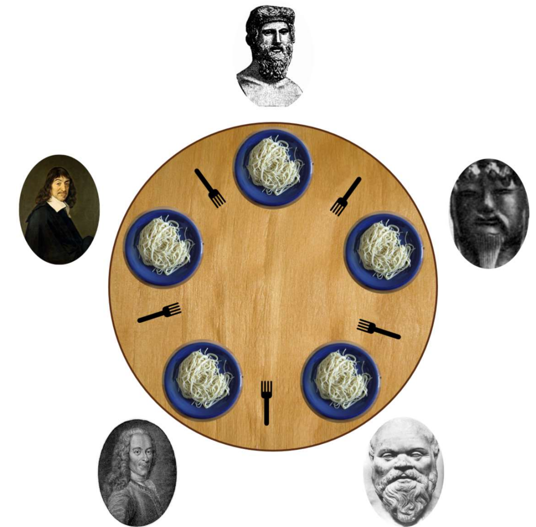
Död låsning (deadlock) kan uppstå om alla filosofer hinner ta upp vänster gaffel samtidigt.

Svält kan uppstå om filosoferna har ett tidscyklist beteende.

Ätande filosofer

Modifiera beteendet/algoritmen så att filosofen lägger tillbaka vänster gaffel inom en slumpmässig tid om inte höger gaffel blir ledig innan dess.

1. Tänk tills vänster gaffel är ledig, sen ta upp den
2. Tänk (slumpmässig tid) tills höger gaffel är ledig, sen ta upp den, annars gå till 5
3. Med båda gafflarna, ät under en bestämd tid
4. Lägg ned höger gaffel
5. Lägg ned vänster gaffel
6. Börja om från början



Död låsning undviks eftersom filosofen ger upp vänster gaffel om man inte kan ta upp den högra. Svält undviks, förutsatt att slumpen inte är tidscyklisk.

Försök till mjukvarulösning av mutual exclusion

```
int owner=0;

//task 0
while (owner!=0) {
  /* do nothing */
}
owner=1;
CRITICAL REGION
owner=0;
```

```
//task 1
while (owner!=0) {
  /* do nothing */
}
owner=1;
CRITICAL REGION
owner=0;
```

Fungerar detta?
Ja, om den ena processen kollar owner ”först”.
Nej, om processerna kollar owner ”samtidigt”. (race condition)

Mjukvarumetod: Peterson’s Algorithm

```
/* global variables */
volatile int flag[2];
volatile int turn;

//task 0
flag[0]=1;
turn=1;
while (flag[1] && turn==1);
CRITICAL REGION
flag[0]=0;
```

```
//task 1
flag[1]=1;
turn=0;
while (flag[0] && turn==0);
CRITICAL REGION
flag[1]=0;
```

Fungerar detta?
Om bara en process begär access (flag[x]=1)? : Ja
Om båda processerna begär access (flag[0,1]=1, turn=0/1? : Ja, eftersom turn kan ju bara vara antingen 0 eller 1

Mjukvarumetod: Peterson’s Algorithm

Notering: Tag bort alla referenser till flag[], dvs använd bara turn. Mutual exclusion fungerar fortfarande men processerna kan då bara använda CRITICAL REGION varannan gång.

Peterson’s algorithm fungerar för två processer, kan expanderas till flera men blir då betydligt mer komplex.

Reality check:

Problem 1: Kompilatorn vet inte att minnespositioner (turn,flag[]) kan ändras av andra processer. Lösning: volatile
Problem 2: Moderna processorer kan ändra på ordningen av läsningar och skrivningar, s k Out of Order Execution (OoOE). Lösning: Använd speciell memory-barrier-instruktion.

Mjukvarumetod: Peterson’s Algorithm

Relevanta begrepp: (googla, kolla wikipedia)

- Sequence points in C/C++ (kontrollpunkter av tidigare evalueringar)
- Out-of-Order Execution (instruktioner i omvänd ordning, effektivare CPU-utnyttj.)
- Memory barrier (minnesskydd p g a OoOE)

Speciella instruktioner:

Assemblerinstruktioner för synkronisering

```
loop:
  TAS (a0)          ; Test And Set: update Z flag, then if a0=0 set a0=1
  BNE loop          ; wait of not set
  CRITICAL REGION
  CLR.B (a0)        ; release lock
```

Synkronisering av processer

Synkronisering behöver göras av olika skäl:

- ”Vanlig” synkronisering:
  För att på ett säkert sätt använda gemensamma resurser

Process A

```
while(1)
  {
    lock M1
    use common resource
    unlock M1
    do other things
  }
```

Process B

```
while(1)
  {
    do something
    lock M1
    use common resource
    unlock M1
  }
```

Synkronisering av processer

Synkronisering behöver göras av olika skäl:

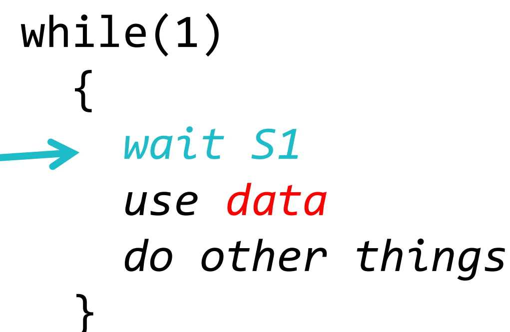
- Asymmetrisk synkronisering:
  För att vänta in en annan process. Lämplig i situationer där processer behöver köra om vartannat, efter varandra (man lämnar över stafettpinnen s a s)

Process A

```
while(1)
  {
    prepare data
    signal S1
    do other things
  }
```

Process B

```
while(1)
  {
    wait S1
    use data
    do other things
  }
```



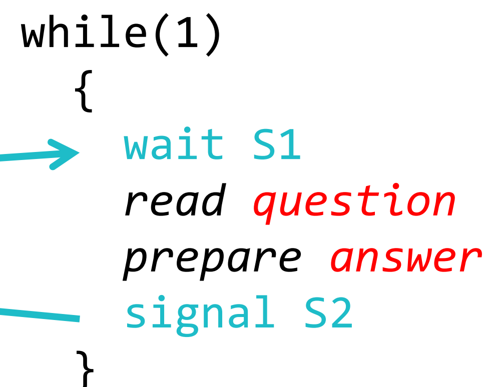
Synkronisering av processer

Synkronisering behöver göras av olika skäl:

- Symmetrisk synkronisering:
  För att utväxla information/data. Två eller flera processer kan samtidigt behöva vara på en specifik plats i koden för att informationsutbyte ska kunna ske.

Process A

```
while(1)
  {
    prepare question
    signal S1
    wait S2
    read answer
  }
```

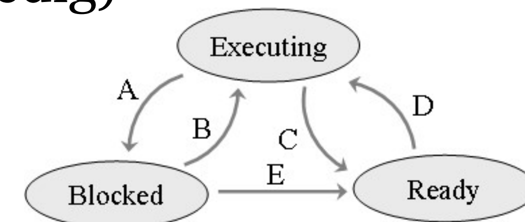


Process B

```
while(1)
  {
    wait S1
    read question
    prepare answer
    signal S2
  }
```

Semaforer

- Har ett värde >= 0
- Initieras till ett visst värde, vanligtvis 1 (gemensam resurs är ledig)
- Har två operationer:
  - Wait: Om semaforens värde > 0, minska värdet med 1, annars vänta, dvs placera processen i väntekö associerad med semaforen: (Executing -> Blocked)
  - Signal: Om ingen process väntar, dvs semaforens väntekö är tom, öka semaforens värde med 1, annars gör någon process ur väntekön körklar: (Blocked -> Ready)

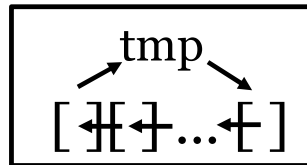


Mutex (”binär semafor”)

- Är antingen Locked eller Unlocked
- Initieras typiskt till att vara Unlocked
- Två operationer:
  -Lock (ungefär ”wait”)
  -Unlock (ungefär ”signal”)

Mutex-exempel

```
// task 1
  for(i=0; i <= 9; i++) {
    array[i] = i;
  }
  // Roterar array[]
  while(1){

    tmp=array[0];

    for(i=0; i <= 8 ; i++){
      array[i]=array[i+1];
    }
    array[9] = tmp;

  }
```




```
// task 2
   // Summera array[]
   while(1){

     sum=0;
     for(i=0; i <= 9; i++){
       sum += array[i];
     }

     printf("Summa: %d\n", sum);
     usleep(1000000);
   }
```

Fråga: Vad kommer task 2 att skriva ut?

`0+1+...+9=45`

[kör programmet]

Mutex-exempel-fixed

```
// task 1
  for(i=0; i <= 9; i++) {
    array[i] = i;
  }
  // Roterar array[]
  while(1){
    pthread_mutex_lock(&Mutex);
    tmp=array[0];

    for(i=0; i <= 8 ; i++){
      array[i]=array[i+1];
    }
    array[9] = tmp;
    pthread_mutex_unlock(&Mutex);
  }
```

```
// task 2
   // Summera array[]
   while(1){
     pthread_mutex_lock(&Mutex);
     sum=0;
     for(i=0; i <= 9; i++){
       sum += array[i];
     }
     pthread_mutex_unlock(&Mutex);
     printf("Summa: %d\n", sum);
     usleep(1000000);
   }
```

Nu blir summan alltid 45!
Eller? (task2 summerar och task1 initierar?)

Mutex-exempel-fixed-asynch

```
// task 1
  for(i=0; i <= 9; i++) {
    array[i] = i;
  }
  sem_post(&Sem);
  // Roterar array[]
  while(1){
    pthread_mutex_lock(&Mutex);
    tmp=array[0];

    for(i=0; i <= 8 ; i++){
      array[i]=array[i+1];
    }
    array[9] = tmp;
    pthread_mutex_unlock(&Mutex);
  }
```

```
// task 2
   sem_wait(&Sem);
   // Summera array[]
   while(1){
     pthread_mutex_lock(&Mutex);
     sum=0;
     for(i=0; i <= 9; i++){
       sum += array[i];
     }
     pthread_mutex_unlock(&Mutex);
     printf("Summa: %d\n", sum);
     usleep(1000000);
   }
```

Nu blir summan alltid 45!
På riktigt!

Semafor: wait (”lock”) i Simple-OS

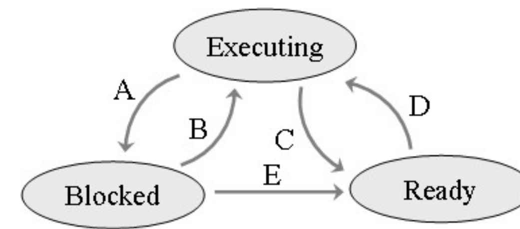
```
void si_sem_wait(si_semaphore *sem)             // wait operation on semaphore sem
{
  int pid;                                      // process id
  DISABLE_INTERRUPTS;                           // atomic section begins

  if (sem->counter > 0) {                       // check counter
    sem->counter--;                             // decrement
  } else {
    pid = process_get_pid_running();            // get pid of running process
    ready_list_remove(pid);                     // remove it from ready list
    wait_list_insert(                           // insert it into the
      sem->wait_list, WAIT_LIST_SIZE, pid);     //   semaphore waiting list
    schedule();                                 // call schedule
  }
  ENABLE_INTERRUPTS;                            // atomic section ends
}
```

Semafor: signal (”unlock”) i Simple-OS

```
void si_sem_signal(si_semaphore *sem)       // signal operation on semaphore sem
{
  int pid;                                  // process id
  DISABLE_INTERRUPTS;                       // atomic section begins
  if (!wait_list_is_empty(                  // check if processes
    sem->wait_list, WAIT_LIST_SIZE))        //   are waiting on semaphore
  {
    pid = wait_list_remove_highest_prio(    // get pid with
      sem->wait_list, WAIT_LIST_SIZE);      //   highest priority
    ready_list_insert(pid);                 // make this process ready to run
    schedule();                             // call schedule
  } else {
    sem->counter++;                         // increment counter
  }
  ENABLE_INTERRUPTS;                        // atomic section ends
}
```

Process : Tillstånd och listor

En process kan befinna sig i olika tillstånd:



En process ligger i någon lista:

Lista	Tillstånd
Ready_List	Ready eller Executing/Running
Wait_List (Semaphore)	Blocked
...	...

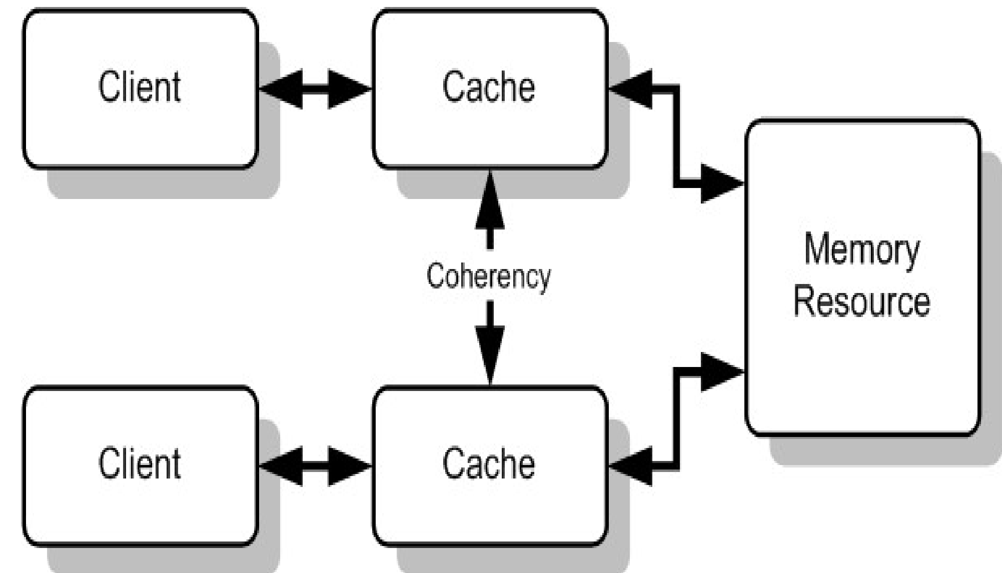
Semafor vs Mutex (förenklad bild)

	Linux	Simple-OS
Semafor	sem_wait count--	si_sem_wait count--
	sem_post count++	si_sem_signal count++
Mutex	pthread_mutex_lock count=0	
	pthread_mutex_unlock count=1	

Semaforer i multicore CPU

```
loop:
→ TAS (a0)          ; Test And Set, upd. Z
  BNE loop          ; wait of not set
  CRITICAL REGION
  CLR.B (a0)        ; release lock
```

```
loop:
→ TAS (a0)          ; Test And Set, upd. Z
  BNE loop          ; wait of not set
  CRITICAL REGION
  CLR.B (a0)        ; release lock
```



Vad händer om TAS körs exakt samtidigt i flera kärnor?

Semaforer i multicore CPU

Via olika tekniker måste processortillverkaren lösa detta, t ex:

- Minnes-lås på bussar och DMA-accesser
  Hårdvaran ger endast access för en process åt gången

- Uppdatering av minne via cache snooping
  Hårdvaruövervakning av cache-minnet för att se vilken process som gör vad

- Olika cache-minnes-tekniker för att uppnå cache coherency
  Hårdvarulösning som säkerställer att olika cache-minnen har samstämmighet

C-programmering

Header files, Makefile, pekare, struct, modularisering

C – header files

Vid C-programmering brukar man skapa s k header-filer, i vilka man placerar programkodens funktionsprototyper, eller funktionshuvuden (därav namnet header file).

```
void foo(int x);
void bar(int y);

void foo(int x)
{
  bar(…);
}

void bar(int y)
{
  foo(…);
}
```

file.h

```
void foo(int x);
void bar(int y);
```

file.c

```
#include ”file.h”

void foo(int x)
{
  bar(…);
}

void bar(int y)
{
  foo(…);
}
```

C – header files

- Eliminerar forwarddeklarationer (bättre inkapsling/abstraktion) från själva programkoden (de ligger ju förstås i header-filen)

- Ger snabbare/bättre programmering och återanvändning av kod
  Kod som delas upp i flera filer/moduler blir mer överblickbar och hanterbar

- Underlättar team work
  Flera programmerare som samtidigt arbetar på samma fil leder lätt till problem

- Ger snabbare kompilering
  Med flera små header-filer måste endast de c-filer som inkluderar en förändrad header-fil kompileras om

C – header files

Inkludering av header-filer är ”i princip” detsamma som att filens innehåll kopieras in på den platsen. Det är typiskt ingen bra ide att kopiera in innehållet, då det lätt leder till ”krockar” med multipla definitioner.

En bra praxis är att alltid ha en header-fil för varje C-fil. T ex till C-filen file-a.c skapar Man en header-fil med namnet file-a.h. Stoppa in alla konstanter, globala variabler och funktionsprototyper i header-filer och inkludera dessa där de behövs.

Egna header-filer inkluderas med

```
#include ”filnamn.h”
```

Systemfiler inkluderas med

```
#include <filnamn.h>
```

C – header files

Makefile

```
proj: file-a.c file-b.c file-a.h file-b.h
	gcc -Wall file-a.c file-b.c -o proj

clean:
	rm -rf proj *~ *.o
```

file-b.h

```
#ifndef FILE_B_H
#define FILE_B_H

int sum(int x, int y);
int square(int x);

#endif
```

file-a.c

```
#include <stdio.h>
#include ”file-a.h”
#include ”file-b.h”

int product(int x, int y)
{
  return x * y;
}

int main(void)
{
  printf("Sum : %d\n", sum(A, B));
  printf("Product : %d\n", product(A, B));
  printf("Square: %d\n", square(A));
  return 0;
}
```

file-a.h

```
#ifndef FILE_A_H
#define FILE_A_H

#define A 7
#define B 4

int product(int x, int y);

#endif
```

file-b.c

```
#include ”file-a.h”
#include ”file-b.h”

int sum(int x, int y)
{
  return x + y;
}

int square(int x)
{
  return product(x, x);
}
```

C - Makefile

Syntaxen för en Makefile:

```
target [target…] : dependency dependency …
--TAB-- command
```

T ex:

```
proj : file-a.c file-b.c file-a.h file-b.h
	gcc –Wall file-a.c file-b.c -o proj
clean:
	rm -rf proj *~ *.o
```

Target proj är till för att bygga projektet. Glöm inte att lägga till eventuellt nyskapade filer som ska ingå.
Target clean är till för att rensa. Dels själva programmet proj, men även tidigare backupfiler (*~) samt objektfiler (*.o).

C - Pointer

```
#include <stdio.h>

int *p, a=5;

Void main() {
  printf(”%x\n”, a);      →5
  printf(”%x\n”, &a);     →6008cc
  printf(”%x\n”, &p);     →6008ca
  p = &a;
  printf(”%x\n”, *p);     →5
  printf(”%x\n”, p);      →6008cc
  *p = 7;
  printf(”%x\n”, a);      →7
}
```

I minnet:

	Adress	Innehåll
	6008c9	
p:	6008ca	6008cc
	6008cb	
a:	6008cc	~~5~~ 7
	6008cd	
	6008ce	

C - Struct

```
struct
{
  int hours;
  int minutes;
  char alarm_text[20];
} clock;

clock.hours = 11;
clock.minutes = 30;
clock.alarm_text[] = ”lunch time”;
```

Struct:ar används för att abstrahera information och för att gruppera sådant som hör ihop.

clock är statiskt allokerad

C – Struct + Pointer

```
typedef struct
{
  int hours;
  int minutes;
  char alarm_text[20];
} clock_data_type;

typedef clock_data_type* clock_type;

clock_type clock;

clock = (clock_type)
  malloc(sizeof(clock_data_type));

clock_set(clock);
```

```
void clock_set(clock_type c)
{
  c->hours = 11;
  c->minutes = 30;
  c->alarm_text[] = ”lunch time”;
}
```

Struct:ens fält refereras nu till via operatorn ->

clock är dynamiskt allokerad

C - Static

```
typedef struct
{
  int hours;
  int minutes;
  char alarm_text[20];
} clock_data_type;

static clock_data_type clock;

void clock_set(void) {
  clock.hours = 11;
  clock.minutes = 30;
  clock.alarm_text[] = ”lunch time”;
}
```

Här blir clock av typen static, vilket medför att den är endast tillgänglig/synlig inom samma fil.

Man tvingar därmed kod som direkt ska påverka variabeln clock att vara i samma fil, och får då bra modularisering.

C – Programming resources

https://www.tutorialspoint.com//cprogramming/index.htm

Pekare och struktar!!

Lab 1 : Introduction, shared resources

Lab 1 : Introduction, shared resources

- ”Installera” Simple-OS
- Modifiera `set_clock.c` : intervall 1:00-12:59, dubbel hastighet
- Använd mutex i `odd_even.c` : så att programmet visar rätt

Ni har tillgång till labsalen Muxen med egna passerkort.
Funkar inte det, skicka mail till mig.

Labmiljö : Terminalprogram, GitLab

Terminalprogram för Windows/Mac/Linux

- https://mobaxterm.mobatek.net/ MobaXterm : Windows
- https://www.cendio.com/thinlinc/download Thinlinc : Windows/Mac/Linux

Anslut till:
`thinlinc.edu.liu.se` (via Thinlinc-klient) eller
`ssh.edu.liu.se` (via ssh-klient, t ex MobaXterm)

Man kan logga in till Muxen utifrån labbet:

```
ssh -XC liuid@muxenx-0yy.ad.liu.se     x={1,2}, yy={01..16}
```

Observera! Stort X, C för komprimering (mindre mängd data)

GitLab : Repositorie för Git

- https://gitlab.liu.se/
- http://www.isy.liu.se/edu/kurs/TSEA81/c_resources.html

Git är bra för att:
- Hantera versioner av program/dokument
- Arbeta med samma program/dokument på flera system
- Samarbeta med andra i samma projekt

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET