Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Semaphore and Monitor

Summer Term 2020 | 07.05.2021

Barry Linnert

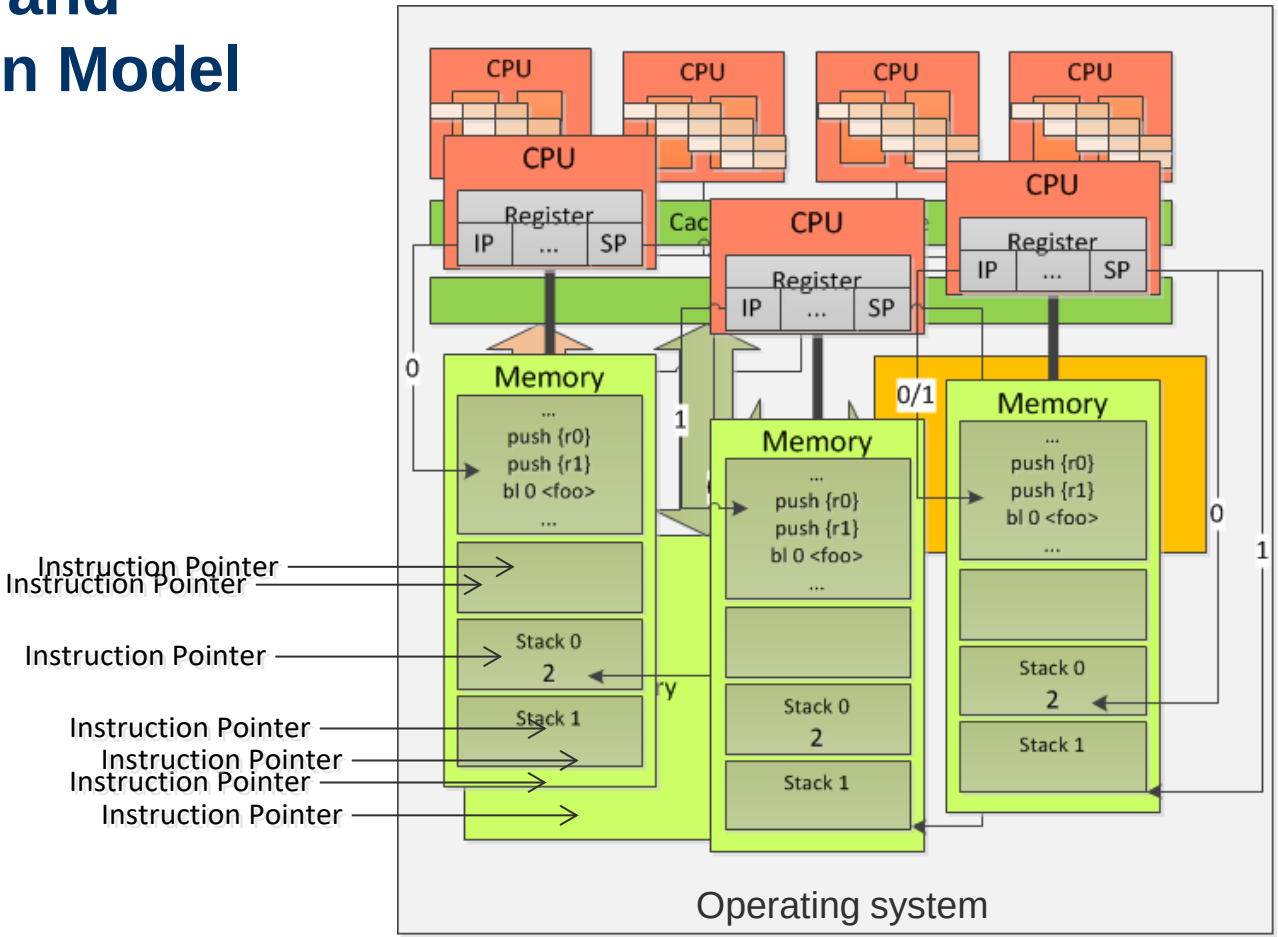
Objectives of Today‘s Lecture

- Semaphore
- Monitors
- Summary

Concepts of Non-sequential and Distributed Programming

SEMAPHORE

Machine and Execution Model



Correctness

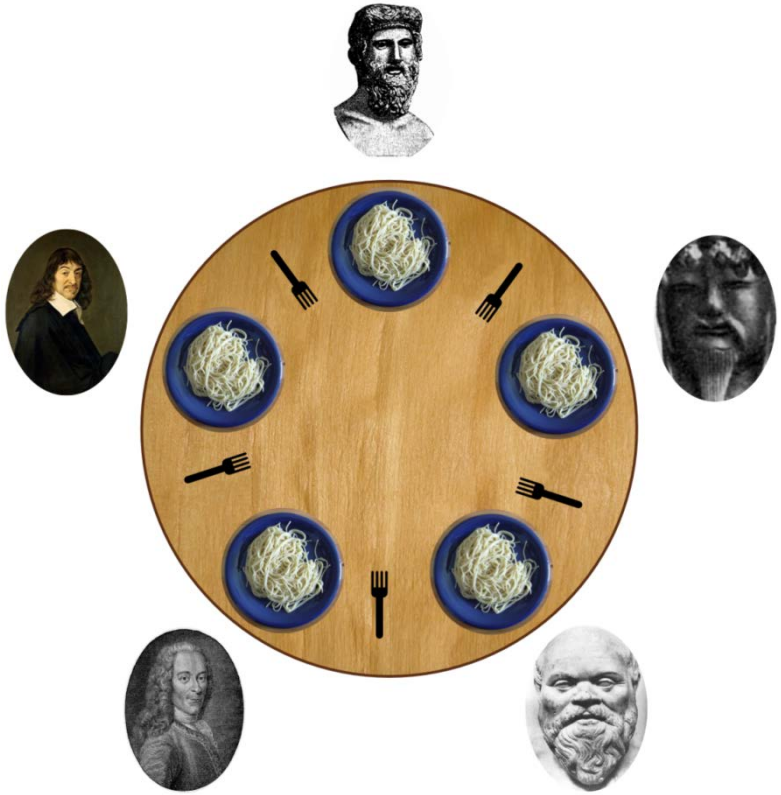
- Correct implementation of commands and functions
  - Compiler/Interpreter, HW
- Correct execution of the set of commands
  - Sequential processing of the operations of the critical section by lock variables (lock/mutex) using the operating system and hardware.
  - Programming model and machine model (execution model) correspond to each other
- Check with
  - Hoare calculation
  - Testing

Requirements for Programs

- The program should do what it has been programmed to do!
  - Functional properties
    - Scope of functions
    - Correctness

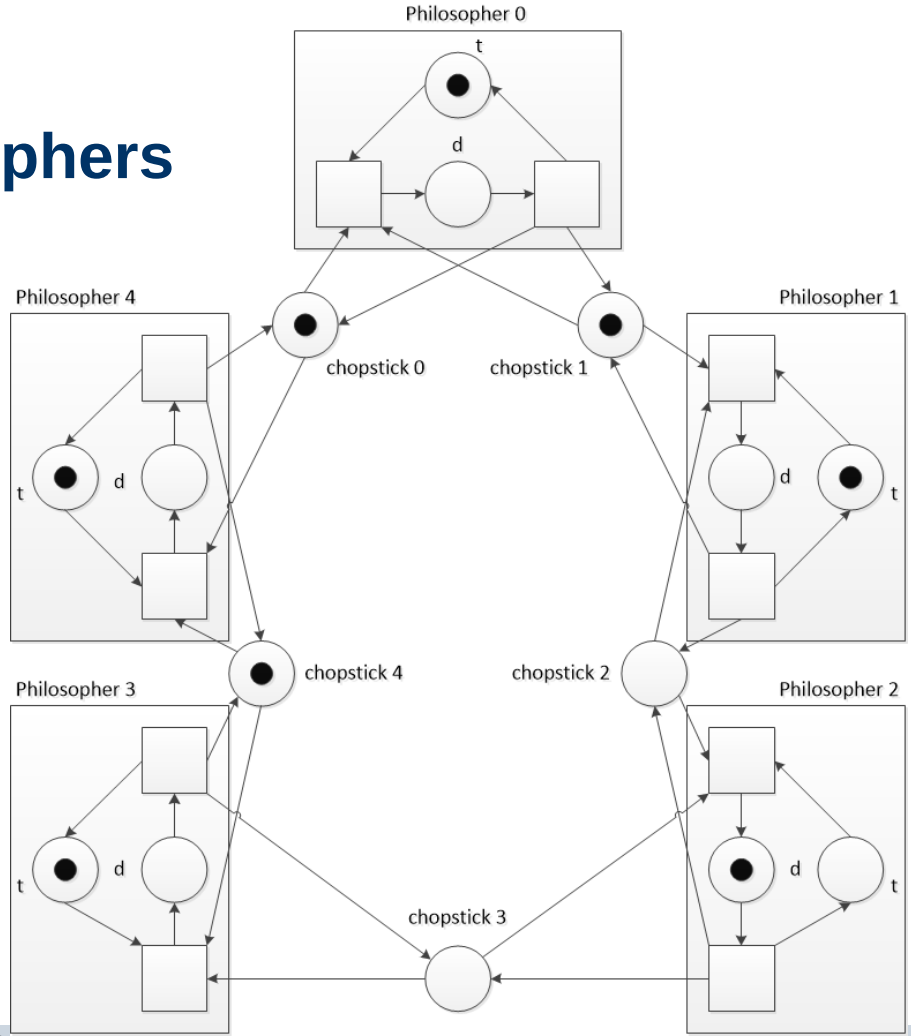
- The program should follow certain rules for this purpose!
  - Non-functional properties
    - Performance
    - Security
    - …

Example: Dining Philosophers

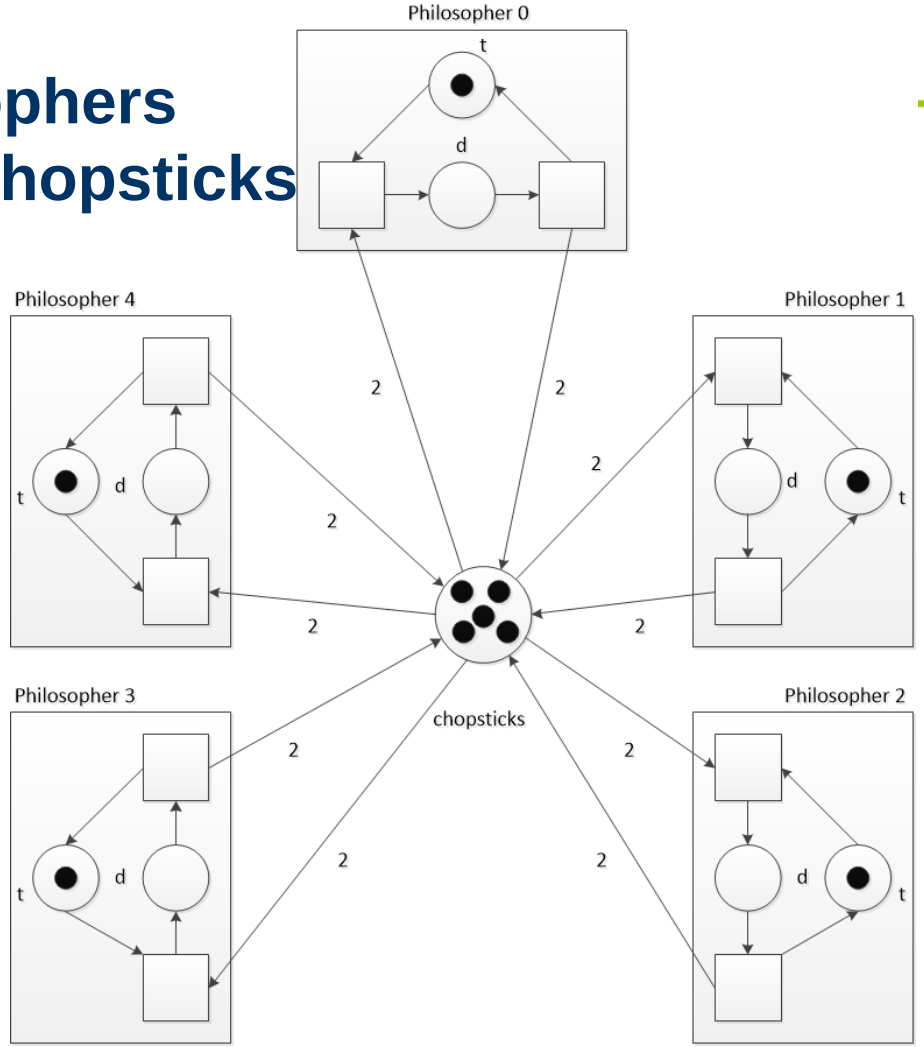


wikipedia.org

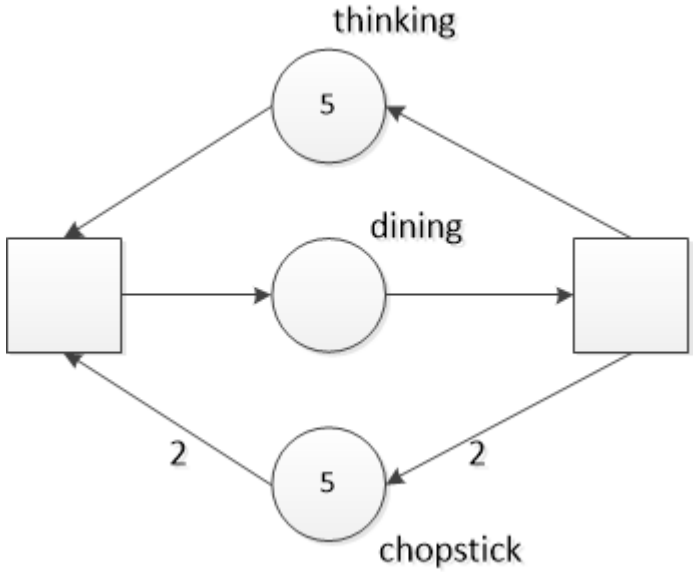
Dining Philosophers



Dining Philosophers with Shared Chopsticks



Dining Philosophers with Shared Chopsticks I



Dining Philosophers with Shared Chopsticks II

- The philosophers share all of the available chopsticks – not just the one beside their plate or bowl.
- It has to be ensured the philosopher equipped with two chopsticks can eat without any interruptions. Thus, the chopsticks should not be snatched by others while in use.
- Hungry philosophers are not able to think and will wait until two chopsticks are available. To reduce the energy consumption of the philosopher (and of the whole system) the waiting philosopher should be resting (sleeping) until the two chopsticks are released.

Dining Philosophers with Shared Chopsticks III

- To implement the approach of the chopstick sharing philosophers we have to deal with some requirements:

- The critical section consists of the check if two chopsticks are available, the picking up of the chopsticks and the process of eating as the chopsticks should not be withdrawn.
- Additionally the resting of the philosopher waiting to get two chopsticks is to be considered. A new state of the philosopher's behavior is to be introduced.

- If enough chopsticks are available, more than one philosopher should be able to eat.

Dining Philosophers with Shared Chopsticks IV

- We use mutual exclusion to protect the critical section:
  - Lock – POSIX `pthread_mutex_trylock()`
  - Resource-saving extension of the lock providing a queue for waiting threads – POSIX `pthread_mutex_lock()`

Is mutual exclusion sufficient to implement the approach of the chopstick sharing philosophers?

- We need a solution protecting the critical section if not enough chopsticks are available and granting access if two or more are free.

Semaphore

- A semaphore is an extension of the lock variable.
- Semaphores are used to protect a critical section, too.
- Their were introduced around 1965 by E.W. Dijkstra
- The term semaphore comes from a flag or signal mast
- Unlike the lock a semaphore is designed to provide access to the critical section for more than one thread or process.
- Thus, it can be seen as a counting lock (counting variable + queue).

wikipedia.org

Semaphore – Operation Mode

- A semaphore works as follows:
- As long as the counter is greater than 0, a thread may enter the critical section.
- If the counter is less or equal 0 the thread is blocked.
- Blocked threads that must not enter the critical section.
- With access of the critical section by a thread the counter is reduced.
- When a thread leaves the critical section the counter is increased.
- The value of the counter represents the number of available resources protected by the critical section.
- If the initial value of the counter equals 1, the semaphore corresponds to a lock variable with a waiting queue.

Semaphore – Functions

- Besides the **semaphore counters the** following operations are required to implement a semaphore:

- **P** – as pass/pack (pass/grasp) - reduces the counter and blocks the thread if the counter <= 0

- **V** – as vrijgeven/verhogen (release or exit/raise) - increments the counter and releases the first blocked thread in the queue (or all blocked threads)

Example Implementation

```
struct semaphore {
    int count;                              // thread counter
    Queue *wt;                              // count>=1: free, count<=0: occupied
}                                           // if count < 0 : |count| is the
                                            // number of waiting threads
void init (semaphore *s, int i) {
    s->count = i;                           // set i=1 for mutual exclusion
    s->wt = NULL;
}
void P (semaphore *s) {
    s->count--;
    if (s->count < 0) block(s->wt);         // enqueue thread
}
void V (semaphore *s) {
    s->count++;
    if (s->count <= 0) deblock(s->wt);      // deblock first of queue
}
```

Source code: 08-00.c

Versions

- Different versions of semaphores can be distinct:

- By the form of the counting variables
  - The simple form with semaphore counter == 1 implements a binary semaphore that resembles a lock with waiting queue.
  - The counting variable can be initialized with any integer value implementing an amount of resources that can be used in parallel at the critical section.

- By the type of managements of threads or processes
  - Implementing active waiting for the threads or
  - block the threads and wake up all or the first after resources are available again.

POSIX semaphores

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

Initialization of the semaphores

```
int sem_wait (sem_t *sem);
```

P - Reduce the counter and block the thread if necessary

```
int sem_post (sem_t *sem);
```

V - increment the counter and release the first thread

POSIX semaphores II

```
int sem_trywait (sem_t *sem);
```

P - Reduce the counter and return without blocking, the return value indicates the success of the attempt to get access to the critical section

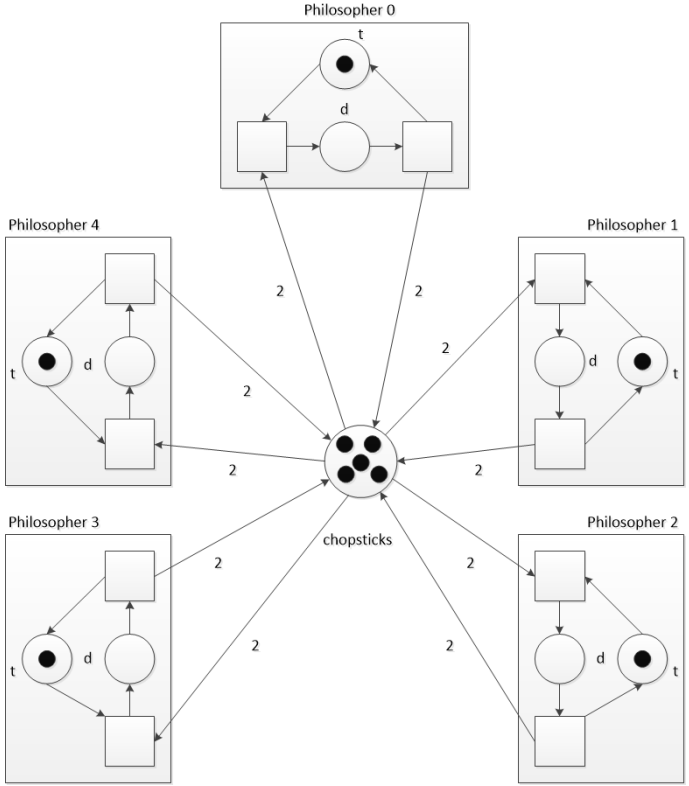
```
int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
```

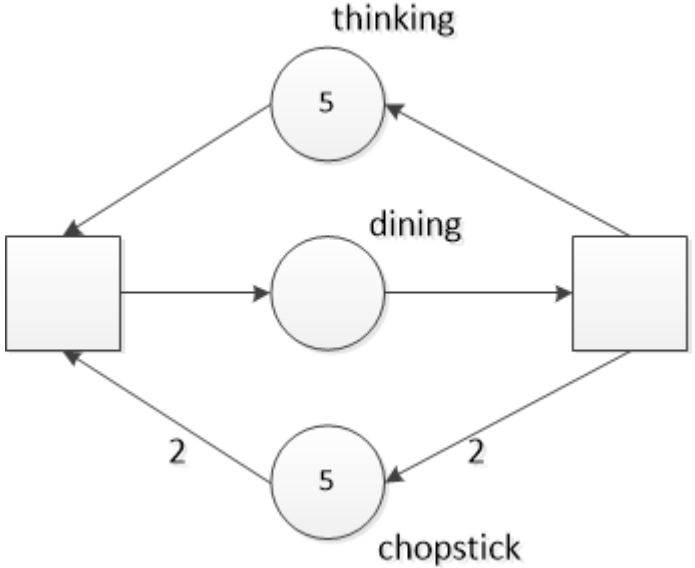
P - Reduce the counter and block the thread if necessary with abort after a given time

```c
// C program to demonstrate working of Semaphores
// https://www.geeksforgeeks.org/use-posix-
// semaphores-c/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;

void* thread(void* arg)
{
  //wait
  sem_wait(&mutex);
  printf("\nEntered..\n");

  //critical section
  sleep(4);

  //signal
  printf("\nJust Exiting...\n");
  sem_post(&mutex);
}

int main()
{
  sem_init(&mutex, 0, 1);
  pthread_t t1,t2;

  pthread_create(&t1,NULL,thread,NULL);
  sleep(2);

  pthread_create(&t2,NULL,thread,NULL);

  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  sem_destroy(&mutex);

  return 0;
}
```

source code: 08-01.c

Dining Philosophers with Shared Chopsticks





```c
// dining philosophers with shared cs
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include <unistd.h>
#define NUM_THREADS     5


void* Philosopher (void *threadid) {
  int i;
  for (i = 0; i < 1000; i++) {
    // thinking
    sleep (2);
    //wait

    printf("\n %d Dining..\n", (long)
        threadid);
    sleep(4);
    printf("\n %d Finished..\n", (long)
        threadid);

  }
}

int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;


  for (t=0; t < NUM_THREADS; t++) {
    rc = pthread_create (&threads[t],
        NULL, Philosopher, (void *)t);
    if (rc) {
      printf ("ERROR; return code from
        pthread_create () is %d\n", rc);
      exit (-1);
    }
  }
  for (t=0; t < NUM_THREADS; t++) {
    pthread_join (threads[t], NULL);
  }

  pthread_exit(NULL);
  return 0;
}
```

source code: 08-02.c

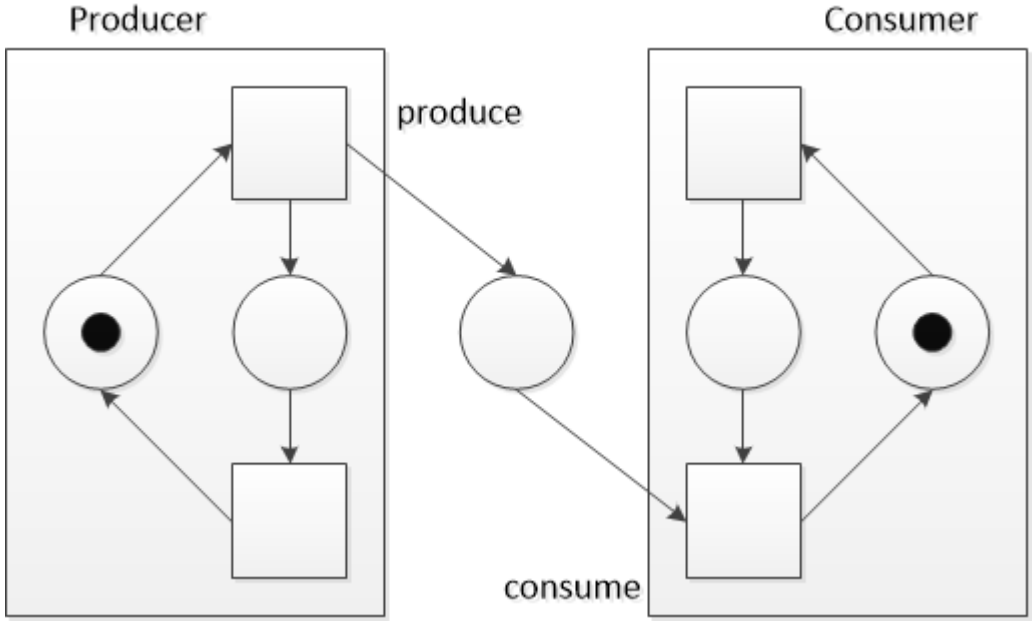
```c
// dining philosophers with shared cs
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS     5
sem_t mutex;

void* Philosopher (void *threadid) {
  int i;
  for (i = 0; i < 1000; i++) {
    // thinking
    sleep (2);
    //wait
    sem_wait(&mutex);
    printf("\n %d Dining..\n", (long)
        threadid);
    sleep(4);
    printf("\n %d Finished..\n", (long)
        threadid);
    sem_post(&mutex);
  }
}

int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;

  sem_init(&mutex, 0, 2); // eating p.
  for (t=0; t < NUM_THREADS; t++) {
    rc = pthread_create (&threads[t],
        NULL, Philosopher, (void *)t);
    if (rc) {
      printf ("ERROR; return code from
        pthread_create () is %d\n", rc);
      exit (-1);
    }
  }
  for (t=0; t < NUM_THREADS; t++) {
    pthread_join (threads[t], NULL);
  }
  sem_destroy(&mutex);
  pthread_exit(NULL);
  return 0;
}
```

source code: 08-02.c

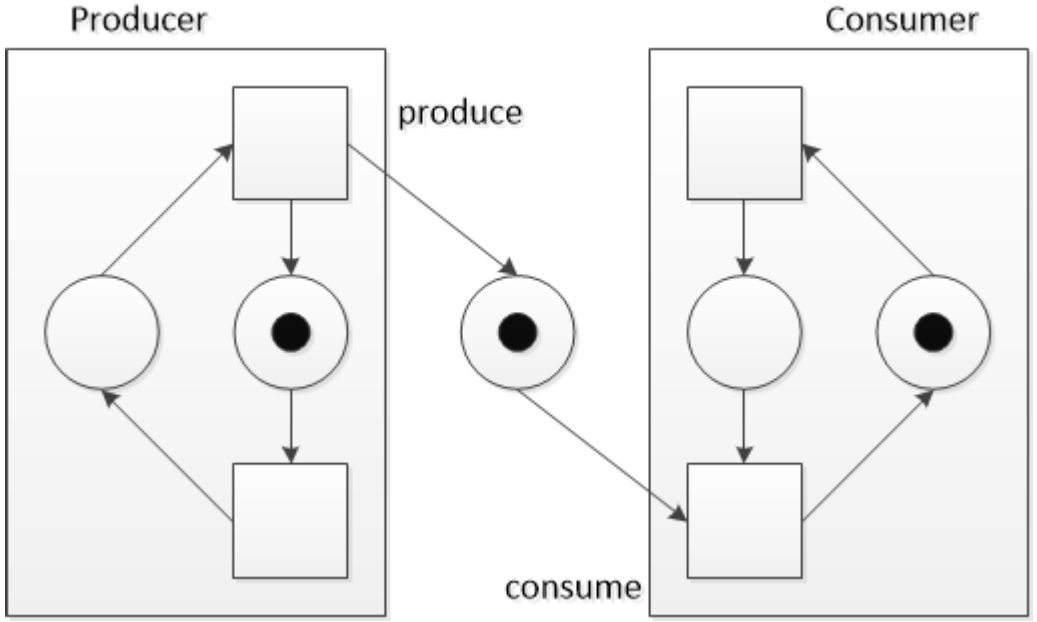
```c
// dining philosophers with shared cs
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS     5
sem_t mutex;

void* Philosopher (void *threadid) {
  int i;
  for (i = 0; i < 1000; i++) {
    // thinking
    sleep (2);
    //wait
    sem_wait(&mutex);
    printf("\n %d Dining..\n", (long)
        threadid);
    sleep(4);
    printf("\n %d Finished..\n", (long)
        threadid);
    sem_post(&mutex);
  }
}

int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;

  sem_init(&mutex, 0, 2); // eating p.
  for (t=0; t < NUM_THREADS; t++) {
    rc = pthread_create (&threads[t],
        NULL, Philosopher, (void *)t);
    if (rc) {
      printf ("ERROR; return code from
        pthread_create () is %d\n", rc);
      exit (-1);
    }
  }
  for (t=0; t < NUM_THREADS; t++) {
    pthread_join (threads[t], NULL);
  }
  sem_destroy(&mutex);
  pthread_exit(NULL);
  return 0;
}
```

source code: 08-02.c

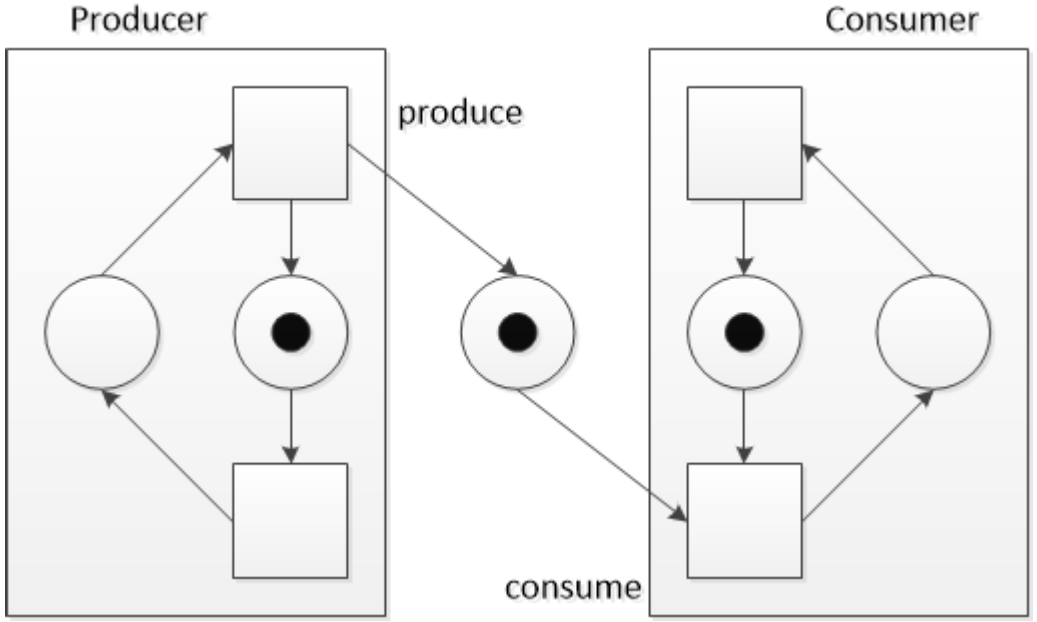
Example: Producer and Consumer



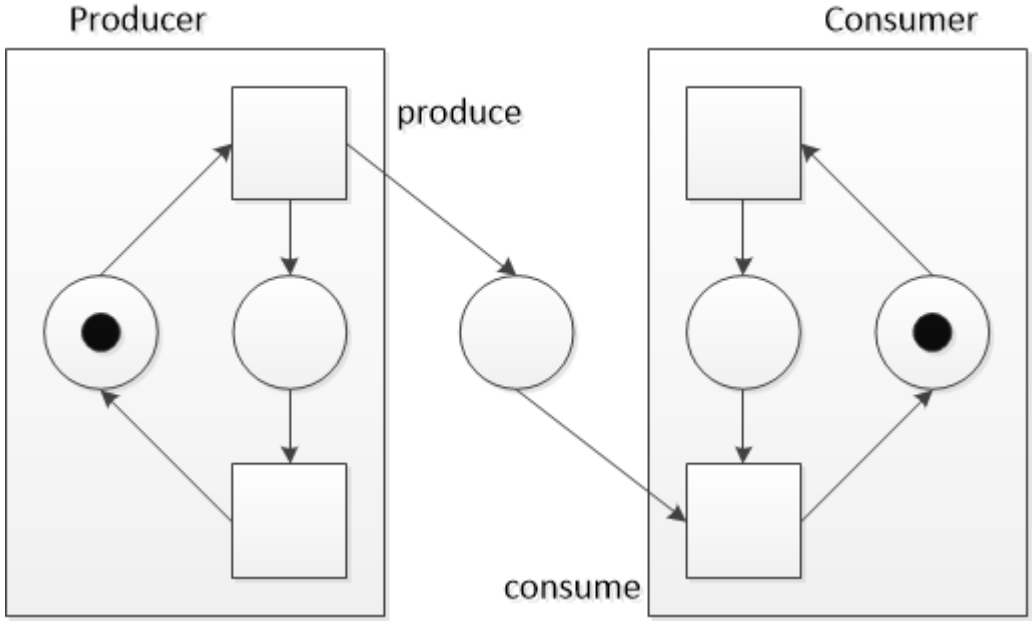
Example: Producer and Consumer



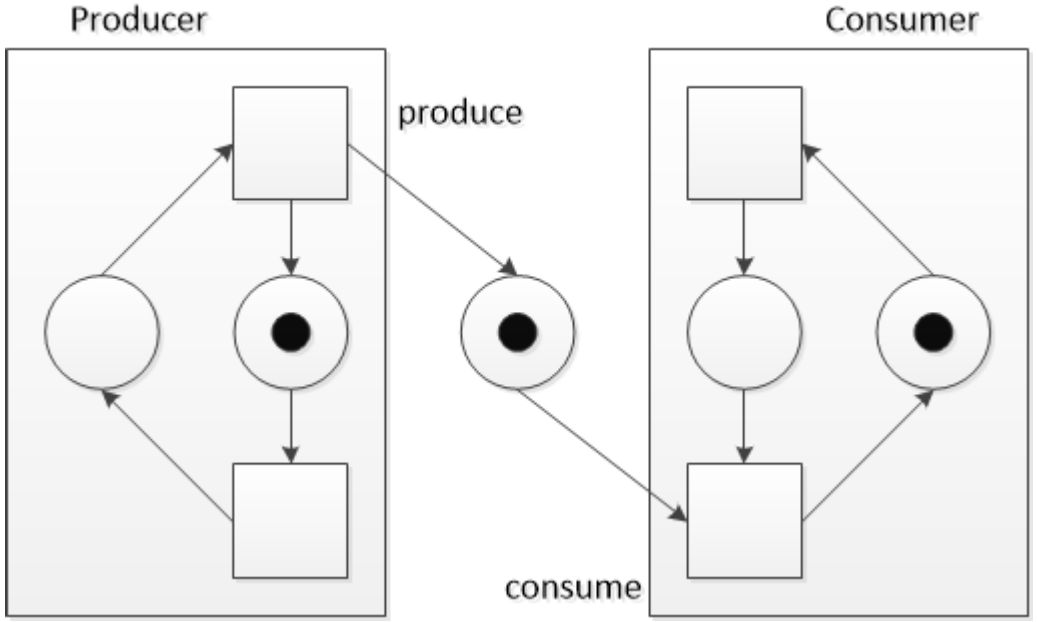
Example: Producer and Consumer



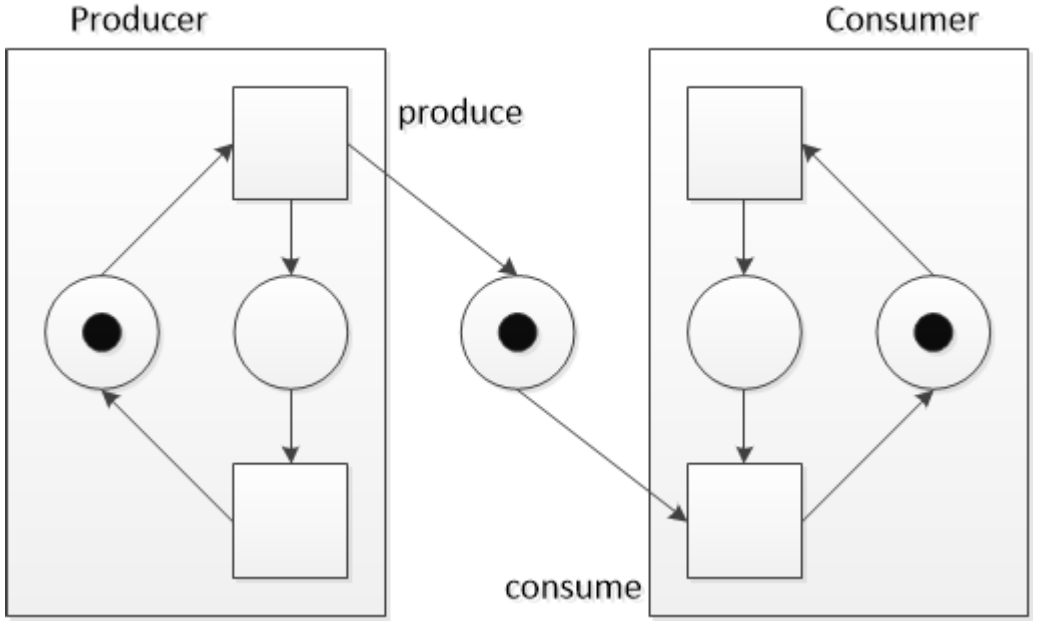
Example: Producer and Consumer



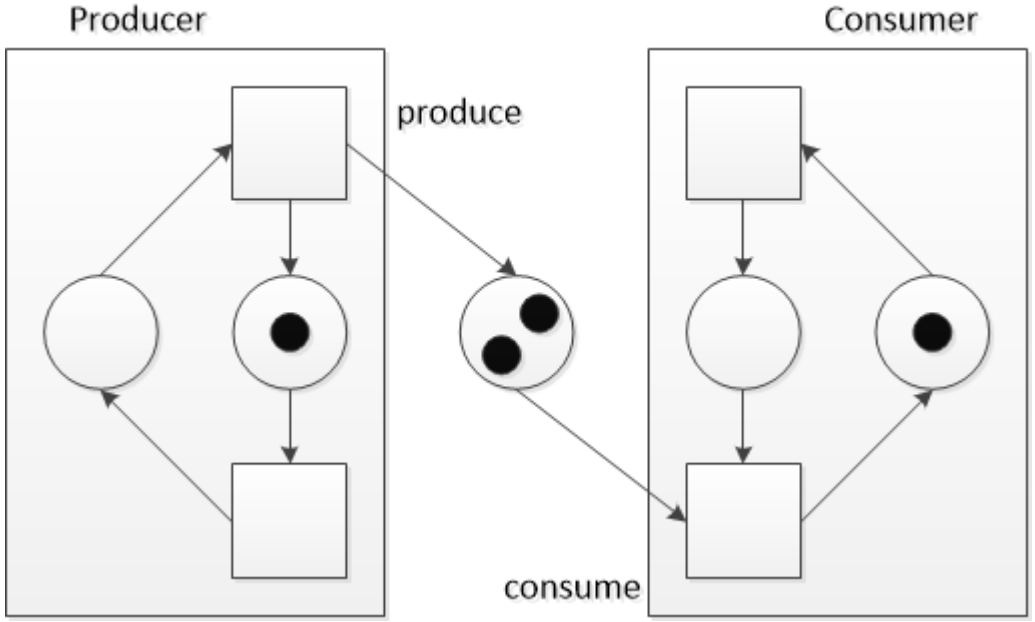
Example: Producer and Consumer



Example: Producer and Consumer



Example: Producer and Consumer



```c
// simple producer consumer example with
// semaphores
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS     5
#define NUM_PLACES      3
sem_t empty; // amount data in buffer
sem_t full; // free places in buffer
sem_t mutex; // critical section
int last;
int buffer[NUM_PLACES];

void* Producer (void *threadid)
{
  ...
}
void* Consumer (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;

  // init semaphores

  // init

  // creating threads

  // joining threads

  // release semaphores

  return 0;
}
```

source code: 08-03.c

```c
// simple producer consumer example with
// semaphores
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS     5
#define NUM_PLACES      3
sem_t empty; // amount data in buffer
sem_t full; // free places in buffer
sem_t mutex; // critical section
int last;
int buffer[NUM_PLACES];

void* Producer (void *threadid)
{
  ...
}
void* Consumer (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[])
{
  ...
  // init all
  // creating threads
  for(t=0; t < NUM_THREADS; t++) {
    if (t == 0)
      rc = pthread_create (&threads[t],
         NULL, Producer, (void *)t);
    else
      rc = pthread_create (&threads[t],
         NULL, Consumer, (void *)t);
     if (rc) {
       exit (-1);
     }
  }
  // joining threads
  for(t=0; t < NUM_THREADS; t++) {
     pthread_join (threads[t], NULL);
  } ...
  pthread_exit(NULL);
}
```

source code: 08-03.c

```c
// simple producer consumer example with
// semaphores
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS     5
#define NUM_PLACES      3
sem_t empty; // amount data in buffer
sem_t full; // free places in buffer
sem_t mutex; // critical section
int last;
int buffer[NUM_PLACES];

void* Producer (void *threadid)
{
  ...
}
void* Consumer (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[])
{
  ...
  // init semaphores
  sem_init(&empty, 0, 0);
  sem_init(&full, 0, 3);
  sem_init(&mutex, 0, 1); // crit. sec.

  // init
  for (t=0; t < NUM_PLACES; t++)
    buffer[t] = 0;
  last = 0;
  // creating threads
  // joining threads

  // release semaphores
  sem_destroy(&mutex);
  sem_destroy(&full);
  sem_destroy(&empty);
  ...
  return 0;
}
```

source code: 08-03.c

```
// simple producer consumer example with
// semaphores
```

```
void* Producer (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {


    buffer[last] = i;
    printf("Producer %d puts %d into
         buffer at place %d \n",
         (long) threadid, buffer[last],
         last);
    last++;


  }
  pthread_exit (NULL);
}
```

```
void* Consumer (void *threadid)
{
  int i;

  while (1) {


    printf ("Consumer %d takes %d \n",
         (long) threadid,
         buffer[last - 1]);
    fflush (stdout);
    last--;


  }
  pthread_exit (NULL);
}
```

source code: 08-03.c

```
// simple producer consumer example with
// semaphores
```

```
void* Producer (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {
    sem_wait(&full);

    buffer[last] = i;
    printf("Producer %d puts %d into
         buffer at place %d \n",
         (long) threadid, buffer[last],
         last);
    last++;


  }
  pthread_exit (NULL);
}
```

```
void* Consumer (void *threadid)
{
  int i;

  while (1) {


    printf ("Consumer %d takes %d \n",
         (long) threadid,
         buffer[last - 1]);
    fflush (stdout);
    last--;

    sem_post(&full);
  }
  pthread_exit (NULL);
}
```

source code: 08-03.c

```
// simple producer consumer example with
// semaphores
```

```
void* Producer (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {
    sem_wait(&full);

    buffer[last] = i;
    printf("Producer %d puts %d into
         buffer at place %d \n",
         (long) threadid, buffer[last],
         last);
    last++;

    sem_post(&empty);
  }
  pthread_exit (NULL);
}
```

```
void* Consumer (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&empty);


    printf ("Consumer %d takes %d \n",
         (long) threadid,
         buffer[last - 1]);
    fflush (stdout);
    last--;

    sem_post(&full);
  }
  pthread_exit (NULL);
}
```

source code: 08-03.c

```
// simple producer consumer example with
// semaphores
```

```
void* Producer (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {
    sem_wait(&full);
    sem_wait(&mutex);
    buffer[last] = i;
    printf("Producer %d puts %d into
         buffer at place %d \n",
         (long) threadid, buffer[last],
         last);
    last++;
    sem_post(&mutex);
    sem_post(&empty);
  }
  pthread_exit (NULL);
}
```

```
void* Consumer (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&empty);
    sem_wait(&mutex);

    printf ("Consumer %d takes %d \n",
         (long) threadid,
         buffer[last - 1]);
    fflush (stdout);
    last--;
    sem_post(&mutex);
    sem_post(&full);
  }
  pthread_exit (NULL);
}
```

source code: 08-03.c

```
// simple producer consumer example with
// semaphores
```

```
void* Producer (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {
    sem_wait(&full);
    sem_wait(&mutex);
    buffer[last] = i;
    printf("Producer %d puts %d into
         buffer at place %d \n",
         (long) threadid, buffer[last],
         last);
    last++;
    sem_post(&mutex);
    sem_post(&empty);
  }
  pthread_exit (NULL);
}
```

```
void* Consumer (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&empty);
    sem_wait(&mutex);

    printf ("Consumer %d takes %d \n",
         (long) threadid,
         buffer[last - 1]);
    fflush (stdout);
    last--;
    sem_post(&mutex);
    sem_post(&full);
  }
  pthread_exit (NULL);
}
```

source code: 08-03.c

Example: Readers and Writers

- Not all of the threads need write access to shared data.
- Read accesses do not change the data and can be executed concurrently or in parallel without any problems.
- Write accesses must be protected (against each other).
- If, in case of a writing thread, read accesses have to be prohibited the situation can be depicted as follows:

	Read	Write
Read	+	–
Write	–	–

Additive Semaphores

- An additive semaphore reduce and increase the counter with values greater than 1 for one thread.

- Thus, the reader-writer problem can be solved by giving the writer exclusive access to the critical section by reduction of the counter to 0 in case of write access.

```c
// simple reader writer example with
// semaphores
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_THREADS     5
#define NUM_WRITERS     2
sem_t writer; // writers in critical sec.
sem_t reader; // readers in critical sec.
int buffer;

void* Writer (void *threadid)
{
  ...
}

void* Reader (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  // init semaphores and buf
  // creating threads
  for (t = 0; t < NUM_THREADS; t++) {
    if (t < NUM_WRITERS)
      rc = pthread_create (&threads[t],
        NULL, Writer, (void *)t);
    else
      rc = pthread_create (&threads[t],
        NULL, Reader, (void *)t);
    if (rc) {
      exit (-1);
    }
  }
  // joining threads
  for (t = 0; t < NUM_THREADS; t++) {
     pthread_join (threads[t], NULL);
  } ...
}
```

source code: 08-04.c

```c
// simple reader writer example with
// semaphores
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_THREADS     5
#define NUM_WRITERS     2
sem_t writer; // writers in critical sec.
sem_t reader; // readers in critical sec.
int buffer;

void* Writer (void *threadid)
{
  ...
}

void* Reader (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;

  // init semaphores and buf
  sem_init(&writer, 0, 1);
  sem_init(&reader, 0, NUM_THREADS -
        NUM_WRITERS);
  buffer = 0;

  // creating threads
  // joining threads

  sem_destroy(&reader);
  sem_destroy(&writer);
  pthread_exit(NULL);

  return 0;
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {


    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);



    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {


    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {



    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);




    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {

    sem_wait(&reader);

    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {



    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);




    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&writer);
    sem_wait(&reader);
    sem_post(&writer);
    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {



    sem_wait(&reader);
    buffer++;
    printf("Writer %d writes %d into
        buffer \n", (long) threadid,
        buffer);


    sem_post(&reader);

    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&writer);
    sem_wait(&reader);
    sem_post(&writer);
    printf ("Reader %d reads %d \n",
        (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {

    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_wait(&reader);
    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);
    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_post(&reader);

    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&writer);
    sem_wait(&reader);
    sem_post(&writer);
    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {
    sem_wait(&writer);
    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_wait(&reader);
    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);
    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_post(&reader);
    sem_post(&writer);
    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&writer);
    sem_wait(&reader);
    sem_post(&writer);
    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

```
// simple reader writer example with
// semaphores

void* Writer (void *threadid)
{
  int i, j;
  for (i = 0; i < 1000; i++) {
    sem_wait(&writer);
    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_wait(&reader);
    buffer++;
    printf("Writer %d writes %d into
         buffer \n", (long) threadid,
         buffer);
    for (j = 0; j < NUM_THREADS -
         NUM_WRITERS; j++)
      sem_post(&reader);
    sem_post(&writer);
    sleep (1);
  }
  pthread_exit (NULL);
}
```

```
void* Reader (void *threadid)
{
  int i;

  while (1) {
    sem_wait(&writer);
    sem_wait(&reader);
    sem_post(&writer);
    printf ("Reader %d reads %d \n",
         (long) threadid, buffer);
    fflush (stdout);



    sem_post(&reader);

  }
  pthread_exit (NULL);
}
```

source code: 08-04.c

Example: Cigarette Smoker‘s Problem

- Three smokers want to smoke and each have a different amount of a specific resource (paper, tobacco, matches).
- None of the participants provides the other with a parts of their own resource.
- The service staff will make one or two of the resources available at irregular intervals (at the earliest after a cigarette length).

- How can everyone smoke at some point?

```c
// simple example of the first attempt to
// solve the cigarette smoker`s problem

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

#define NUM_THREADS     4
sem_t   resource[3];
sem_t   finished;
int tids[NUM_THREADS];

void* Service (void *threadid)
{
  ...
}

void* Smoker (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;
  // init semaphores
  // creating threads
  for (i = 0; i < NUM_THREADS; i++) {
    tids[i] = i;
    if (i == 0)
      rc = pthread_create (&threads[i],
        NULL, Service, (void*) &tids[i]);
    else
      rc = pthread_create (&threads[i],
        NULL, Smoker, (void*) &tids[i]);
    if (rc) {
      exit (-1);
    }
  }
  // joining threads
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join (threads[i], NULL);
  }
```

source code-03-05.c

```c
// simple example of the first attempt to
// solve the cigarette smoker`s problem

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

#define NUM_THREADS     4
sem_t   resource[3];
sem_t   finished;
int tids[NUM_THREADS];

void* Service (void *threadid)
{
  ...
}

void* Smoker (void *threadid)
{
  ...
}
```

```c
int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;

  // init semaphores
  sem_init(&finished, 0, 1);
  sem_init(&resource[0], 0, 0);
  sem_init(&resource[1], 0, 0);
  sem_init(&resource[2], 0, 0);

  srand ((unsigned) time (NULL));
  // creating threads
  // joining threads

  sem_destroy(&finished);
  sem_destroy(&resource[0]);
  sem_destroy(&resource[1]);
  sem_destroy(&resource[2]);
  pthread_exit (NULL);
  return 0;
}
```

source code: 08-05.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Service (void *threadid)
{
  int i, random_nr;
  for (i= 0; i < 1000; i++) {



    if (random_nr < RAND_MAX / 3) {


    }
    if ((random_nr >= RAND_MAX / 3)&&(random_nr < 2 * (RAND_MAX / 3))) {


    }
    if (random_nr >= 2 * (RAND_MAX / 3)) {


    }
  }
  pthread_exit (NULL);
}
```

source code: 08-05.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Service (void *threadid)
{
  int i, random_nr;
  for (i= 0; i < 1000; i++) {
    sem_wait (&finished);


    if (random_nr < RAND_MAX / 3) {


    }
    if ((random_nr >= RAND_MAX / 3)&&(random_nr < 2 * (RAND_MAX / 3))) {


    }
    if (random_nr >= 2 * (RAND_MAX / 3)) {


    }
  }
  pthread_exit (NULL);
}
```

source code: 08-05.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Service (void *threadid)
{
  int i, random_nr;
  for (i= 0; i < 1000; i++) {
    sem_wait (&finished);
    random_nr = rand ();

    if (random_nr < RAND_MAX / 3) {


    }
    if ((random_nr >= RAND_MAX / 3)&&(random_nr < 2 * (RAND_MAX / 3))) {


    }
    if (random_nr >= 2 * (RAND_MAX / 3)) {


    }
  }
  pthread_exit (NULL);
}
```

source code: 08-05.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Service (void *threadid)
{
  int i, random_nr;
  for (i= 0; i < 1000; i++) {
    sem_wait (&finished);
    random_nr = rand ();

    if (random_nr < RAND_MAX / 3) {
      sem_post(&resource[0]);
      sem_post(&resource[1]);
    }
    if ((random_nr >= RAND_MAX / 3)&&(random_nr < 2 * (RAND_MAX / 3))) {
      sem_post(&resource[0]);
      sem_post(&resource[2]);
    }
    if (random_nr >= 2 * (RAND_MAX / 3)) {
      sem_post(&resource[1]);
      sem_post(&resource[2]);
    }
  }
  pthread_exit (NULL);
}
```

source code: 08-05.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Smoker (void *threadid)
{
  int i;
  for (i= 0; i < 333; i++) {
    printf ("Smoker %d waits\n", *((int *) threadid));
    if (*((int *) threadid) == 1) {
      sem_wait (&resource[0]); printf ("Smoker %d got 0 \n", *((int *) threadid));
      sem_wait (&resource[1]); printf ("Smoker %d got 1 \n", *((int *) threadid));
    }
    if (*((int *) threadid) == 2) {
      sem_wait (&resource[1]); printf ("Smoker %d got 1 \n", *((int *) threadid));
      sem_wait (&resource[2]); printf ("Smoker %d got 2 \n", *((int *) threadid));
    }
    if (*((int *) threadid) == 3) {
      sem_wait (&resource[0]); printf ("Smoker %d got 0 \n", *((int *) threadid));
      sem_wait (&resource[2]); printf ("Smoker %d got 2 \n", *((int *) threadid));
    }
    printf ("Smoker %d is smoking\n", *((int *) threadid));
    sem_post(&finished);
  }
  pthread_exit (NULL);
}
```

source code: 08-05.c

Example: Cigarette Smoker‘s Problem – Approach

- Combine the resources to build a new resource.
- A separate thread deals with management of the resources and the combination of the resources.

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Service (void *threadid)
{
  int i, random_nr;
  for (i= 0; i < 1000; i++) {
    sem_wait (&finished);
    random_nr = rand ();
    if (random_nr < RAND_MAX / 3) {
      res[0]++;
      res[1]++;
    }
    if ((random_nr >= RAND_MAX / 3)&&(random_nr < 2 * (RAND_MAX / 3))) {
      res[0]++;
      res[2]++;
    }
    if (random_nr >= 2 * (RAND_MAX / 3)) {
      res[1]++;
      res[2]++;
    }
    sem_post (&table);
  }
  pthread_exit(NULL);
}
```

source code: 08-06.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Help (void *threadid)
{
  int i;

  for (i= 0; i < 1000; i++) {
    sem_wait (&table);
    if ((res[0] > 0)&&(res[1] > 0)) {
      res[0]--; res[1]--;
      sem_post (&res_0_1);
    }
    if ((res[0] > 0)&&(res[2] > 0)) {
      res[0]--; res[2]--;
      sem_post (&res_0_2);
    }
    if ((res[1] > 0)&&(res[2] > 0)) {
      res[1]--; res[2]--;
      sem_post (&res_1_2);
    }
  }
  pthread_exit(NULL);
}
```

source code: 08-06.c

```
// simple example of the first attempt to solve the cigarette smoker`s problem

void* Smoker (void *threadid)
{
  int i;

  for (i= 0; i < 333; i++) {
    printf ("Smoker %d waits\n", *((int *) threadid));
    if (*((int *) threadid) == 1) {
      sem_wait (&res_0_1); printf("Smoker %d got 0 and 1 \n",*((int *) threadid));
    }
    if (*((int *) threadid) == 2) {
      sem_wait (&res_1_2); printf("Smoker %d got 1 and 2 \n",*((int *) threadid));
    }
    if (*((int *) threadid) == 3) {
      sem_wait (&res_0_2); printf("Smoker %d got 0 and 2 \n",*((int *) threadid));
    }

    printf ("Smoker %d is smoking\n", *((int *) threadid));
    sem_post(&finished);
  }
  pthread_exit(NULL);
}
```

source code: 08-06.c

Concepts of Non-sequential and Distributed Programming

MONITOR

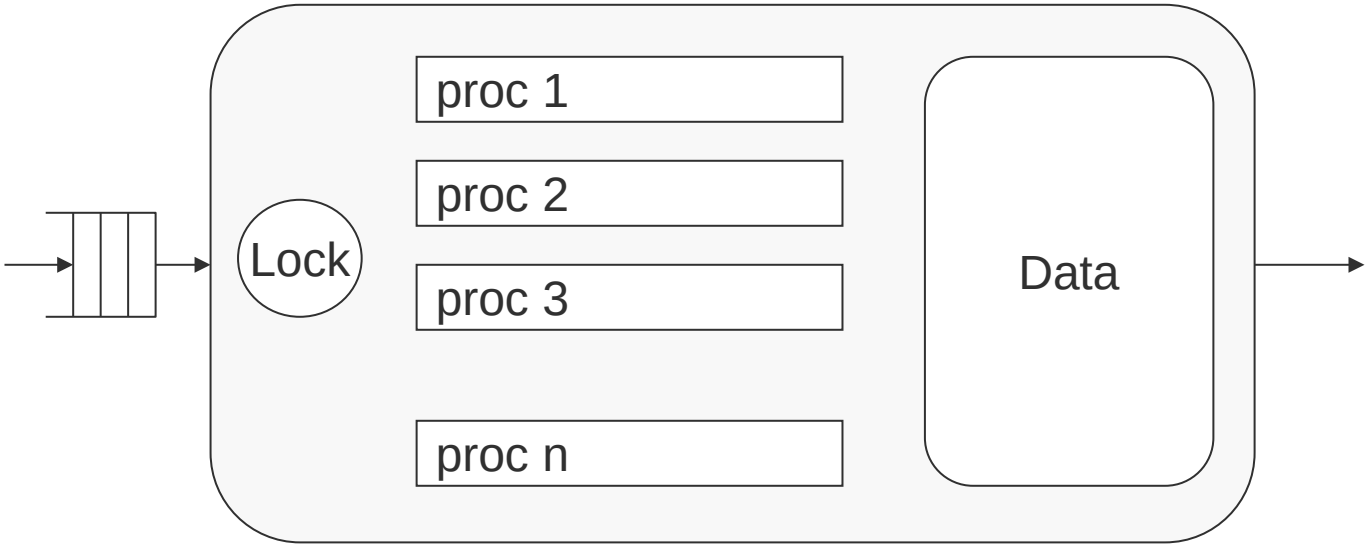
Monitor

- The usage of locks to manage the access to a critical section is error-prone.
- A safer solution would be an automatic lock and release for access to shared data.
- An object that guarantees mutual exclusion without requiring the programmer to explicitly insert lock and unlock operations is called **monitor**.
- A monitor is an object consisting of procedures (methods/functions) and data structures that ensures that at any time it is used by not more than one thread.
- Thus, a monitor is an extension of a lock or semaphore.

- The concept of a monitor was introduced by C.A.R. Hoare in 1974.

- The operating system kernel itself represents a monitor.

Monitor II



Condition variables

- Blocking a thread within the monitor blocks the entire monitor (even the functions that are not affected).
- While a process is waiting for a condition variable, the monitor must be released for other processes.
- Two operations are provided to realize the condition synchronization via the condition variable:
  - **cwait(c)** process releases monitor and waits for the following csignal(c), i.e. the occurrence of condition c. It then continues in the monitor. The process is blocked in any case!
  - **csignal(c)** A waiting process is released. The monitor is occupied again. If there is no waiting process, the procedure has no effect.
- The waiting processes are managed in a queue (as with signaling or semaphores).

POSIX condition variables

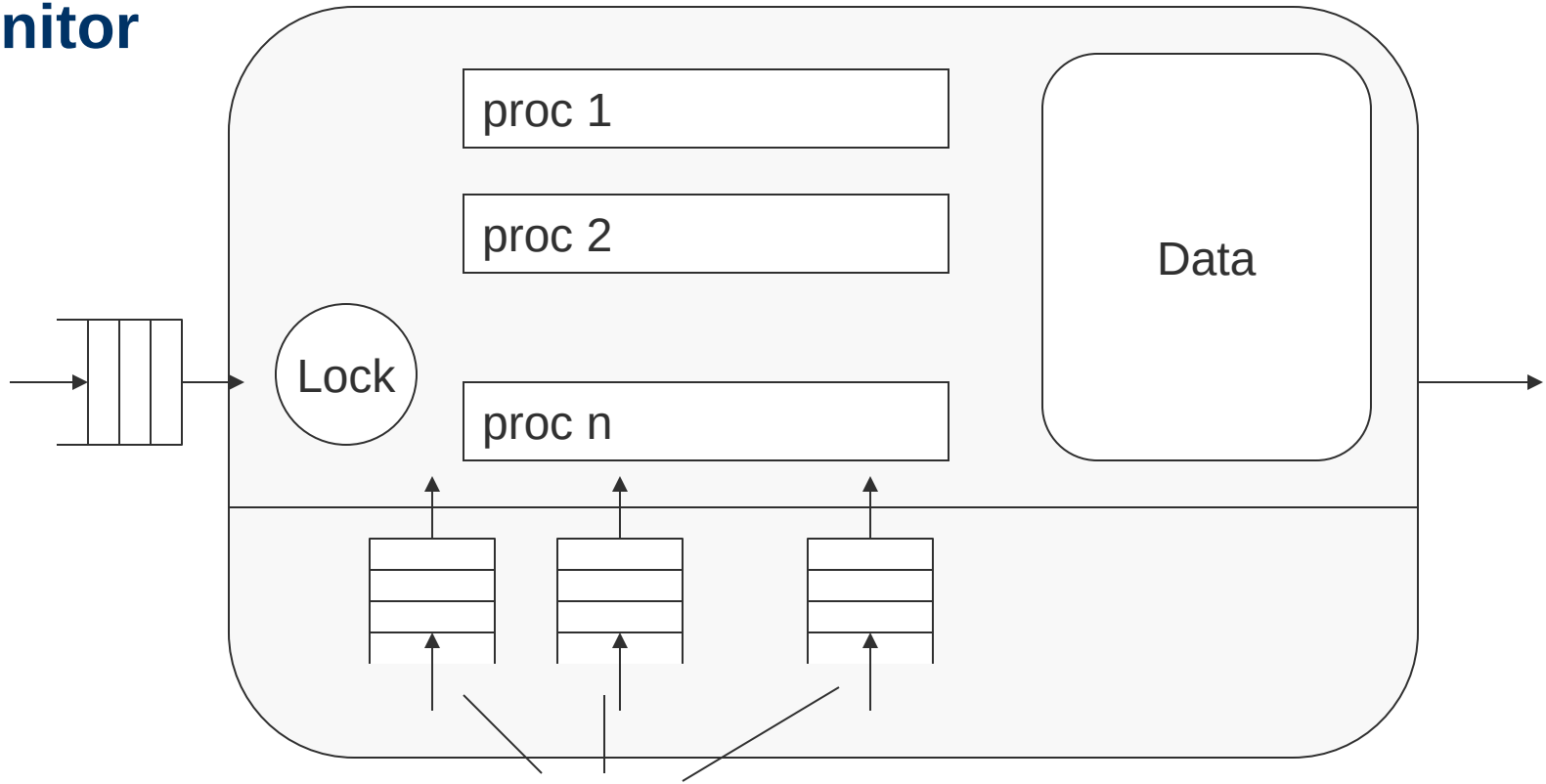
```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex);
```

Waiting for condition `cond` and release of mutex `mutex`

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex,
       const struct timespec *restrict abstime);
```

Waiting for condition `cond` and release of mutex `mutex` with interruption after `abstime`

Monitor



Queues for waiting for the condition variables

Concepts of Non-sequential and Distributed Programming

SUMMARY

Summary – Programming and Correctness

- We program our programs using high level programming languages as it makes it easier to abstract from the specific machine, write code for different hardware architectures and are able to focus on the problem and algorithm to solve the problem.
- The operations of the high level programming language are translated to machine instructions.
- The programming follows a model (programming model) that has to be transferred reliably to the corresponding execution and machine model, which is the basis for the execution of the machine instructions.
- If the transfer from the programming model – with the specific program as one example – to the execution and machine model that determines the execution of the program is correct, the execution of the program will be correct.

Summary – Determinism

- To cover the transfer by the example of the program the terms *determinism, deterministic program or algorithm* and *deterministic execution* of the program were introduced.
- *Deterministic program and execution* imply that given a specific input the program will always produce the same specific output and will always fulfill changes to the state of the system in the same way (and order).
- Thus, the deterministic execution of the program is based on executing the same machine instructions in exactly the same order or sequence (using the same input and generating the same output data).

Summary – Processes

- Sequential programming and execution of the program as just described are well known from previous courses.
- Unfortunately, they come with some drawbacks. One is the un-efficient usage of resources as the programs have to be executed one after the other (first-in-first-out/first-come-first-served).
- The solution for the problem combining the conflicting requirements of deterministic execution and the efficient usage of the machine was the introduction of processes.
- Thus, the frame to apply the term *determinism* was adapted from the whole machine to the abstraction or virtualization represented by the process executing the program.

Summary – Threads

- The requirements for a deterministic execution of the program are fulfilled even if the process is blocked and has to wait to resume it’s execution after execution of other processes.
- As the concept of processes comes with separated address spaces the interchange of data is difficult.
- To overcome the problem and to have different parts of the program use the same data, threads were introduced.
- These execution paths work together in one address spaces on shared data and therefore enable easy data interchange for programs running on one machine.
- Unfortunately, the term *determinism* cannot be used anymore as the deterministic execution of threads cannot be enforced and the concept *thread* does not provide a secure frame as shared data overcome the borders.

Summary – Critical Section

- To deal with this problem the term determinism was displaced by the term *determined algorithm* or *determined program (execution)*.
- This should be sufficient as the result of the execution is the interesting part and the order of execution of the instruction (and operations of the programming language) is only the vehicle to enforce the correct execution.
- But beware: the usage of shared data may lead to unexpected program behavior.
- These parts of the program are described as critical sections.
- These critical sections have to be protected! This protection enforces a deterministic execution and usage of the data and therefore the correct execution based on determined execution of the program.

Summary – Protection of the Critical Section

- The approaches to protect the critical sections – locks, semaphores, monitors – and their algorithms usually are critical sections or use operations representing critical sections themselves.
- Therefore, some of these approaches rely on specific characteristics of the machine depict by the machine model.
- And of course, if the machine architecture is not consistent with the model (or vice versa) the approaches will not protect the critical section reliably.

Requirements for Solutions to protect the Critical Section

- The solution has to protect the critical section reliably by mutual exclusion.
- The solution should be used in higher level programming languages.
  - Thus, the solution is usable on different architectures providing portability for the program using it.
- The solution must not lead to a deadlock.
- The solution should provide low overhead.
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair.

Literature

- Herrtwich, R.G.; Hommel, G.: *Kooperation und Konkurrenz – Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*. Studienreihe Informatik, Springer-Verlag, 1989
- Gregory R. Andrews: *Foundations of Multithreaded, Parallel, and Distributed Programming*, Pearson, 2000
- Maurer, Christian: *Nichtsequentielle und Verteilte Programmierung mit Go*, Springer-Verlag, 2018
- Thomas Rauber, Gudula Rünger: *Parallele und verteilte Programmierung*, Springer-Verlag, 2000
- Michael J. Quinn: *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Science, 2003

Concepts of Non-sequential and Distributed Programming

NEXT LECTURE

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

OpenMP

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2021)