Computação em Larga Escala

Problem of the producers and the consumers – Algorithmic analysis

António Rui Borges

Summary

- *Problem formulation*
- *Synchronization*
- *Solution with monitors*

Problem formulation - 1

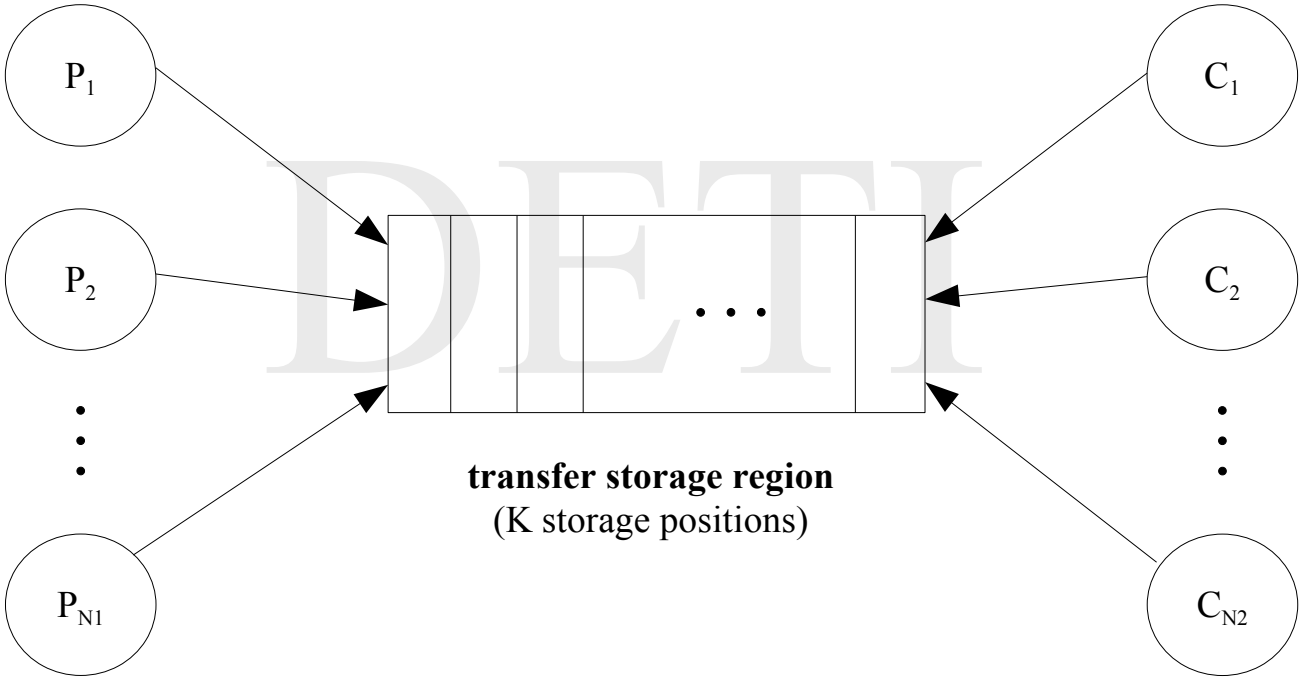
The *problem of the producers and the consumers* is a problem that is traditionally used to evaluate the functionality of new synchronization mechanisms in concurrent programming.

There are two classes of entities

- the *producers*, which execute an infinite loop where they produce data of some sort, stored it in a shared region and do something else
- the *consumers*, which also execute an infinite loop where they retrieve data from the shared region, consume it in some way and do something else.

The interaction among them must proceed in an organized manner, independently of the number of producers and consumers, the time they take to fulfill they loop activities and the storage capacity of the shared region.

Problem formulation - 2



Problem formulation - 3

Producer life cycle

```
forever
{ produceVal (&data);
  putVal (data);
  doSomethingElse ();
}
```

Consumer life cycle

```
forever
{ getVal (&data);
  consumeVal (data);
  doSomethingElse ();
}
```

Synchronization

Since multiple activities, of type `putVal` and `getVal`, are taking place concurrently over the transfer region, one must avoid racing conditions which will lead to inconsistency of information, deadlock or starvation.

Thus, one must ensure that

- *mutual exclusion* – only a single `putVal` or `getVal` operation may be executed at a time
- *producer synchronization point* – when the transfer region is full, the *producers* can not store data and must wait for a location to be empty
- *consumer synchronization point* – when the transfer region is empty, the *consumers* can not retrieve data and must wait for a location to be occupied
- every *producer* must be able to store sooner or later data in the transfer region
- every consumer should be able to retrieve sooner or later data from the transfer region
- every piece of data, stored in the transfer region, should be sooner or later retrieved.

Solution with monitors

A correct solution to the problem can be obtained by transforming the transfer region into a monitor. If the code which implements the solution is written in C, one can use the functionalities provided by the library `pthread` to do this.

In this sense, one has

- *mutual exclusion* – through the calling of the operations of `lock` and `unlock` on a *mutex* variable
- *producer synchronization point* – through the calling of the operations of `wait` (to block) and `signal` (to wake up) on a *condition* variable
- *consumer synchronization point* – through the calling of the operations of `wait` (to block) and `signal` (to wake up) on a *condition* variable.

FIFO as a monitor - 1

Internal data structure

```
/** \brief producer threads return status array */
extern int statusProd[N];

/** \brief consumer threads return status array */
extern int statusCons[N];

/** \brief storage region */
static unsigned int mem[K];

/** \brief insertion pointer */
static unsigned int ii;

/** \brief retrieval pointer */
static unsigned int ri;

/** \brief flag signaling the data transfer region is full */
static bool full;

/** \brief locking flag which warrants mutual exclusion inside the monitor */
static pthread_mutex_t accessCR = PTHREAD_MUTEX_INITIALIZER;

/** \brief flag which warrants that the data transfer region is initialized exactly once */
static pthread_once_t init = PTHREAD_ONCE_INIT;;

/** \brief producers synchronization point when the data transfer region is full */
static pthread_cond_t fifoFull;

/** \brief consumers synchronization point when the data transfer region is empty */
static pthread_cond_t fifoEmpty;
```

FIFO as a monitor - 2

Initialization of the internal data structure


```
/**
 *  \brief Initialization of the data transfer region.
 *
 *  Internal monitor operation.
 */

static void initialization (void)
{
                                                          /* initialize FIFO in empty state */
  ii = ri = 0;                                  /* FIFO insertion and retrieval pointers set to the same value */
  full = false;                                                          /* FIFO is not full */

  pthread_cond_init (&fifoFull, NULL);                        /* initialize producers synchronization point */
  pthread_cond_init (&fifoEmpty, NULL);                       /* initialize consumers synchronization point */
}
```

FIFO as a monitor - 3

Access primitive

```
/**
 *  \brief Store a value in the data transfer region.
 *         Operation carried out by the producers.
 *              \param prodId producer identification
 *              \param val value to be stored
 */

void putVal (unsigned int prodId, unsigned int val)
{
  if ((statusProd[prodId] = pthread_mutex_lock (&accessCR)) != 0)                    /* enter monitor */
     { errno = statusProd[prodId];                                                   /* save error in errno */
       perror ("error on entering monitor(CF)");
       statusProd[prodId] = EXIT_FAILURE;
       pthread_exit (&statusProd[prodId]);
     }
  pthread_once (&init, initialization);                                  /* internal data initialization */
  while (full)                                                  /* wait if the data transfer region is full */
  { if ((statusProd[prodId] = pthread_cond_wait (&fifoFull, &accessCR)) != 0)
       { errno = statusProd[prodId];                                                /* save error in errno */
         perror ("error on waiting in fifoFull");
         statusProd[prodId] = EXIT_FAILURE;
         pthread_exit (&statusProd[prodId]);
       }
  }
  mem[ii] = val;                                                          /* store value in the FIFO */
  ii = (ii + 1) % K;
  full = (ii == ri);
  if ((statusProd[prodId] = pthread_cond_signal (&fifoEmpty)) != 0)      /* let a consumer know that a value has
                                                                                                been stored */
     { errno = statusProd[prodId];                                                  /* save error in errno */
       perror ("error on signaling in fifoEmpty");
       statusProd[prodId] = EXIT_FAILURE;
       pthread_exit (&statusProd[prodId]);
     }
  if ((statusProd[prodId] = pthread_mutex_unlock (&accessCR)) != 0)                  /* exit monitor */
     { errno = statusProd[prodId];                                                  /* save error in errno */
       perror ("error on exiting monitor(CF)");
       statusProd[prodId] = EXIT_FAILURE;
       pthread_exit (&statusProd[prodId]);
     }
}
```

FIFO as a monitor - 4

Access primitive

```
/**
 *  \brief Get a value from the data transfer region.
 *         Operation carried out by the consumers.
 *              \param consId consumer identification
 *              \return value
 */

unsigned int getVal (unsigned int consId)
{
  unsigned int val;                                                    /* retrieved value */

  if ((statusCons[consId] = pthread_mutex_lock (&accessCR)) != 0)          /* enter monitor */
     { errno = statusCons[consId];                                   /* save error in errno */
       perror ("error on entering monitor(CF)");
       statusCons[consId] = EXIT_FAILURE;
       pthread_exit (&statusCons[consId]);
     }
  pthread_once (&init, initialization);                          /* internal data initialization */
  while ((ii == ri) && !full)                          /* wait if the data transfer region is empty */
  { if ((statusCons[consId] = pthread_cond_wait (&fifoEmpty, &accessCR)) != 0)
       { errno = statusCons[consId];                                 /* save error in errno */
         perror ("error on waiting in fifoEmpty");
         statusCons[consId] = EXIT_FAILURE;
         pthread_exit (&statusCons[consId]);
       }
  }
  val = mem[ri];                                              /* retrieve a  value from the FIFO */
  ri = (ri + 1) % K;
  full = false;
  if ((statusCons[consId] = pthread_cond_signal (&fifoFull)) != 0)      /* let a producer know that a value has
                                                                                    been retrieved */
     { errno = statusCons[consId];                                   /* save error in errno */
       perror ("error on signaling in fifoFull");
       statusCons[consId] = EXIT_FAILURE;
       pthread_exit (&statusCons[consId]);
     }
  if ((statusCons[consId] = pthread_mutex_unlock (&accessCR)) != 0)        /* exit monitor */
     { errno = statusCons[consId];                                   /* save error in errno */
       perror ("error on exiting monitor(CF)");
       statusCons[consId] = EXIT_FAILURE;
       pthread_exit (&statusCons[consId]);
     }
  return val;
}
```

Generator thread - 1

```
/** \brief producer threads return status array */
int statusProd[N];

/** \brief consumer threads return status array */
int statusCons[N];

/** \brief producer life cycle routine */
static void *producer (void *id);

/** \brief consumer life cycle routine */
static void *consumer (void *id);

/**
 *  \brief Main thread.
 *
 *  Its role is starting the simulation by generating the intervening entities threads (producers and consumers)
 *  and waiting for their termination.
 */

int main (void)
{
  pthread_t tIdProd[N],                                          /* producers internal thread id array */
            tIdCons[N];                                          /* consumers internal thread id array */
  unsigned int prod[N],                                  /* producers application defined thread id array */
               cons[N];                                  /* consumers application defined thread id array */
  int i;                                                                        /* counting variable */
  int *status_p;                                                      /* pointer to execution status */
```

Generator thread - 2

```
  /* initializing the application defined thread id arrays for the producers and the consumers and the random
     number generator */
  double t0, t1;                                                                    /* time limits */
  for (i = 0; i < N; i++)
    prod[i] = i;
  for (i = 0; i < N; i++)
    cons[i] = i;
  srandom ((unsigned int) getpid ());
  t0 = ((double) clock ()) / CLOCKS_PER_SEC;

  /* generation of intervening entities threads */
  for (i = 0; i < N; i++)
    if (pthread_create (&tIdProd[i], NULL, producer, &prod[i]) != 0)                /* thread producer */
       { perror ("error on creating thread producer");
         exit (EXIT_FAILURE);
       }
  for (i = 0; i < N; i++)
    if (pthread_create (&tIdCons[i], NULL, consumer, &cons[i]) != 0)                /* thread consumer */
       { perror ("error on creating thread consumer");
         exit (EXIT_FAILURE);
       }

  /* waiting for the termination of the intervening entities threads */
  printf ("\nFinal report\n");
  for (i = 0; i < N; i++)
  { if (pthread_join (tIdProd[i], (void *) &status_p) != 0)                         /* thread producer */
       { perror ("error on waiting for thread producer");
         exit (EXIT_FAILURE);
       }
    printf ("thread producer, with id %u, has terminated: ", i);
    printf ("its status was %d\n", *status_p);
  }
  for (i = 0; i < N; i++)
  { if (pthread_join (tIdCons[i], (void *) &status_p) != 0)                         /* thread consumer */
       { perror ("error on waiting for thread customer");
         exit (EXIT_FAILURE);
       }
    printf ("thread consumer, with id %u, has terminated: ", i);
    printf ("its status was %d\n", *status_p);
  }
  t1 = ((double) clock ()) / CLOCKS_PER_SEC;
  printf ("\nElapsed time = %.6f s\n", t1 - t0);

  exit (EXIT_SUCCESS);
}
```

Generator thread - 3

Thread Producer

```
/**
 *  \brief Function producer.
 *
 *         Its role is to simulate the life cycle of a producer.
 *
 *              \param par pointer to application defined producer identification
 */

static void *producer (void *par)
{
  unsigned int id = *((unsigned int *) par),                                  /* producer id */
               val;                                                         /* produced value */
  int i;                                                                  /* counting variable */

  for (i = 0; i < M; i++)
  { val = 1000 * id + i;                                                    /* produce a value */
    putVal (id, val);                                                        /* store a value */
    usleep((unsigned int) floor (40.0 * random () / RAND_MAX + 1.5));      /* do something else */
  }

  statusProd[id] = EXIT_SUCCESS;
  pthread_exit (&statusProd[id]);
}
```

Generator thread - 4

Thread Consumer

```
/**
 *  \brief Function consumer.
 *
 *         Its role is to simulate the life cycle of a consumer.
 *
 *              \param par pointer to application defined consumer identification
 */

static void *consumer (void *par)
{
  unsigned int id = *((unsigned int *) par),                                    /* consumer id */
               val;                                                           /* produced value */
  int i;                                                                     /* counting variable */

  for (i = 0; i < M; i++)
  { usleep((unsigned int) floor (40.0 * random () / RAND_MAX + 1.5));         /* do something else */
    val = getVal (id);                                                        /* retrieve a value */
    printf ("The value %u was produced by the thread P%u and consumed by the thread C%u.\n",          /* consume
                                                                                                       a value */
             val % 1000, val / 1000, id);
  }

  statusCons[id] = EXIT_SUCCESS;
  pthread_exit (&statusCons[id]);
}
```

Compiling and linking the code

```
gcc -Wall -O3 -o producersConsumers producersConsumers.c fifo.c -lpthread -lm
```