DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

Lab 6

Process Synchronization

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

Synchronization of processes using IPC

In computer science, inter-process communication (IPC) refers to the set of mechanisms that an operating system provides to allow the user processes to manage **shared data**.

There can be several methods to communicate among process:

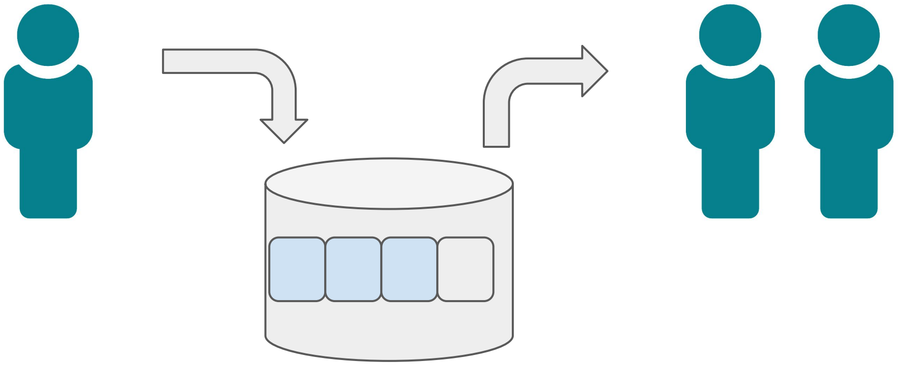
- sockets (we will see them)
- pipes (standard IO)
- signals
- message queues
- semaphores
- shared memory

Here we are focusing on which of them can help us to synchronize our processes during the concurrent execution.

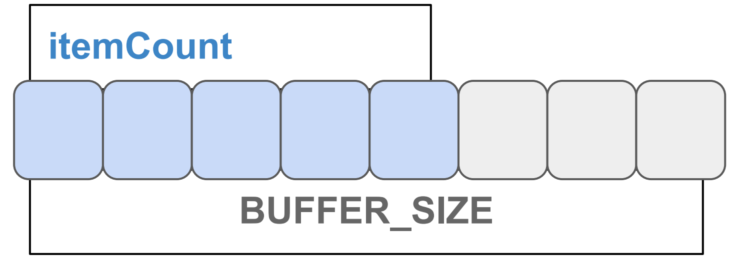
The producer-consumer example

A typical example of a multi-process synchronization problem

The **producer** repeatedly generates data and writes it into the buffer. At the same time the **consumer** independently reads the data in the buffer, removing it in the course of reading it, and using that data in some way.



Example 1




```
int itemCount = 0;
```

Process 1

```
void producer() {
    int item = 0;
    while (1) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) { sleep(); }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) { wakeup(consumer); }
    }
}
```

Process 2

```
void consumer() {
    int item;
    while (1) {
        if (itemCount == 0) { sleep(); }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) { wakeup(producer); }
        consumeItem(item);
    }
}
```

Race Azard: Deadlock !

at start **itemCount = 0**

Process 1

```
void producer() {
   int item = 0;
   while (1) {
```

context switch

```
      item = produceItem();
      if (itemCount == BUFFER_SIZE) { sleep(); }
      putItemIntoBuffer(item);
      itemCount = itemCount + 1;
      if (itemCount == 1) {
         wakeup(consumer); }
```

context switch

```
   }
}
```

Process 2

```
void consumer() {
   int item;
   while (1) {
      if (itemCount == 0) {
```

```
         sleep(); }
      item = removeItemFromBuffer();
      itemCount = itemCount - 1;
      if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer); }
      consumeItem(item);
   }
}
```

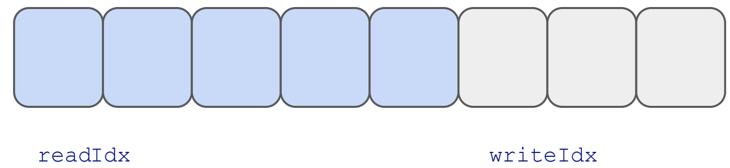
Example 2


```
struct BufferData {
 int readIdx;
 int writeIdx;
 int buffer[BUFFER_SIZE];
};

int next(int id) { return (id+1)%BUFFER_SIZE; }

/* producer routine */
static void producer() {
 int item = 0;
 while(1) {
   if(sharedBuf->readIdx != sharedBuf->writeIdx) {
   /* Write data item */
   sharedBuf->buffer[sharedBuf->writeIdx] = item;
   /* Update write index */
   sharedBuf->writeIdx = next(sharedBuf->writeIdx);
 }}
}

/* Consumer routine */
static void consumer() {
 int item;
 while(1) {
  if(next(sharedBuf->writeIdx) != sharedBuf->readIdx) {
   /* Get data item */
   item = sharedBuf->buffer[sharedBuf->readIdx];
   /* Update read index */
   sharedBuf->readIdx = next(sharedBuf->readIdx);
 }}
}
```



Race Condition

CODING EXAMPLE

CODING EXAMPLE

```
/* producer routine */
static void producer() {
 int item = 0;
 while(1) {


   if(sharedBuf->readIdx != sharedBuf->writeIdx) {
   /* Write data item */
   sharedBuf->buffer[sharedBuf->writeIdx] = item;
   /* Update write index */
   sharedBuf->writeIdx = next(sharedBuf->writeIdx);

 }}
}
```

```
/* Consumer routine */
static void consumer() {
 int item;
 while(1) {
  if(next(sharedBuf->writeIdx)






!= sharedBuf->readIdx) {
   /* Get data item */
   item = sharedBuf->buffer[sharedBuf->readIdx];
   /* Update read index */
   sharedBuf->readIdx = next(sharedBuf->readIdx);
 }}
}
```

Semaphores

Linux semaphores are counting semaphores and are widely used to synchronize processes. When a semaphore has been created and an initial value assigned, two operations can be performed on it: **sem_wait()** and **sem_post()**.

There are two kinds of semaphores in Linux:

a. **named:** associated with a name (character string)

   created with:

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

b. **unnamed:**

   created with:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Semaphores example

The following example is an implementation the producer/consumer application where the producer and the consumers execute on different processes and use unnamed semaphores to manage the critical section and to handle synchronization.

In particular, the initial value of the semaphore (mutexSem) used to manage the critical section is set to one, thus ensuring that **only one process at a time can enter the critical section** by issuing first a sem_wait() and then a sem_post() operation.

The other two semaphores (dataAvailableSem and roomAvailableSem) will contain the current number of available data slots and free ones, respectively.

EXAMPLE 1: Semaphores

CODING EXAMPLE

CODING EXAMPLE

```
struct BufferData {
 int readIdx;
 int writeIdx;
 int buffer[BUFFER_SIZE];
 sem_t mutexSem;
 sem_t dataAvailableSem;
 sem_t roomAvailableSem;
};

struct BufferData *sharedBuf;




/* Consumer routine */
static void consumer() {
 int item;
 while(1) {


/* Wait for availability of at least one data slot */
   sem_wait(&sharedBuf->dataAvailableSem);
/* Enter critical section */
   sem_wait(&sharedBuf->mutexSem);
/* Get data item */
   item = sharedBuf->buffer[sharedBuf->readIdx];
/* Update read index */
   sharedBuf->readIdx = (sharedBuf->readIdx + 1)%BUFFER_SIZE;
/* Signal that a new empty slot is available */
   sem_post(&sharedBuf->roomAvailableSem);
/* Exit critical section */
   sem_post(&sharedBuf->mutexSem);
/* Consume data item and take actions (e.g return)*/
  // ...
 }
}
```

```
/* producer routine */
static void producer() {
 int item = 0;
 while(1)  {
/* Produce data item and take actions (e.g. return)*/
   // ...
/* Wait for availability of at least one empty slot */
   sem_wait(&sharedBuf->roomAvailableSem);
/* Enter critical section */
   sem_wait(&sharedBuf->mutexSem);
/* Write data item */
   sharedBuf->buffer[sharedBuf->writeIdx] = item;
/* Update write index */
   sharedBuf->writeIdx = (sharedBuf->writeIdx + 1)%BUFFER_SIZE;
/* Signal that a new data slot is available */
   sem_post(&sharedBuf->dataAvailableSem);
/* Exit critical section */
   sem_post(&sharedBuf->mutexSem);
 }
}
```

EXAMPLE 1: Semaphores

CODING EXAMPLE

CODING EXAMPLE

```c
int main(int argc, char *args[])
{
 int memId;
 int i, nConsumers;
 pid_t pids[MAX_PROCESSES];

 sem_init(&sharedBuf->mutexSem, 1, 1);
 sem_init(&sharedBuf->dataAvailableSem, 1, 0);
 sem_init(&sharedBuf->roomAvailableSem, 1, BUFFER_SIZE);

/* Launch producer process */
 pids[0] = fork();
 if(pids[0] == 0) {
/* Child process */
   producer();
   exit(0);
 }
/* Launch consumer processes */
 for(i = 0; i < nConsumers; i++) {
   pids[i+1] = fork();
   if(pids[i+1] == 0)
   {
     consumer();
     exit(0);
   }
 }
/* Wait process termination */
 for(i = 0; i <= nConsumers; i++) {
   waitpid(pids[i], NULL, 0);
 }
 return 0;
}
```

Message Queues

The message queue example is much simpler than the previous one because there is no need to actually worry about synchronization at all !! Everything is managed by the operating system as a stream of data flowing from producer to consumer, and all the synchronization is handled within the kernel queue implementation.

We will see other means of synchronization that uses message passing with sockets.

Example2: Message Queue

CODING EXAMPLE

CODING EXAMPLE

```
/* Message structure definition */
struct msgbuf {
 long mtype;
 int item;
};

int msgId;

/* Consumer routine */
static void consumer() {
 int retSize;
 struct msgbuf msg;
 int item;
 while(1)  {
/* Receive the message. msgrcv returns the size of the received message */
   retSize =  msgrcv(msgId, &msg, sizeof(int), PRODCONS_TYPE, 0);
   if(retSize == -1)  {  perror("error msgrcv"); exit(0); }
   item = msg.item;
/* Consume data item */
   // ...
 }
}
/* Consumer routine */
static void producer()  {
 int item = 0;
 struct msgbuf msg;
 msg.mtype = PRODCONS_TYPE;
 while(1)  {
/* produce data item */
   // ...
   msg.item = item;
   msgsnd(msgId, &msg, sizeof(int), 0);
 }
}
```

Example2: Message Queue

CODING EXAMPLE

CODING EXAMPLE

```
int main(int argc, char *args[]) {
 int i, nConsumers;
 pid_t pids[MAX_PROCESSES];
 if(argc != 2) {
   printf("Usage: prodcons  <nConsumers>\n");
   exit(0);
 }
 sscanf(args[1], "%d", &nConsumers);
/* Initialize message queue */
 msgId = msgget(IPC_PRIVATE, 0666);
 if(msgId == -1) {
   perror("msgget");
   exit(0);
 }
/* Launch producer process */
 pids[0] = fork();
 if(pids[0] == 0) {
/* Child process */
   producer();
   exit(0);
 }
/* Launch consumer processes */
 for(i = 0; i < nConsumers; i++) {
   pids[i+1] = fork();
   if(pids[i+1] == 0)   {
     consumer();
     exit(0);
   }
 }
/* Wait process termination */
 for(i = 0; i <= nConsumers; i++) {
   waitpid(pids[i], NULL, 0);
 }
 return 0;
```

POSIX Threads synchronization

Threads synchronization happens with mutexes and condition variables in a code pattern also called **Monitor**.

Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

1) MUTEX LOCK

The operating system will put any thread trying to lock an already locked mutex in wait state, and such threads will be made ready as soon as the mutex is unlocked.

2) CONDITION WAIT

Condition variables can be used only by the methods of the monitor they belong to, and cannot be referenced in any way from outside the monitor boundary.

Condition variables

CODING EXAMPLE

CODING EXAMPLE

```
/* The mutex used to protect shared data */
pthread_mutex_t mutex;

/* Condition variables to signal availability of room and data in the buffer */
pthread_cond_t roomAvailable, dataAvailable;

#define BUFFER_SIZE 128
int buffer[BUFFER_SIZE];
int readIdx = 0;
int writeIdx = 0;

/* Producer code. Passed argument is not used */
static void *producer(void *arg) {
 int item = 0;
 while(1) {
/* Produce a new item and take actions (e.g. return) */
   //  ...
/* Enter critical section */
   pthread_mutex_lock(&mutex);
/* Wait for room availability */
   while((writeIdx + 1)%BUFFER_SIZE == readIdx)    {
     pthread_cond_wait(&roomAvailable, &mutex);
   }
/* At this point room is available Put the item in the buffer */
   buffer[writeIdx] = item;
   writeIdx = (writeIdx + 1)%BUFFER_SIZE;
/* Signal data avilability */
   pthread_cond_signal(&dataAvailable);
/* Exit critical section */
   pthread_mutex_unlock(&mutex);
 }
 return NULL;
}
```

Condition variables

CODING EXAMPLE

CODING EXAMPLE

```c
/* Consumer Code: the passed argument is not used */
static void *consumer(void *arg)
{
 int item;
 while(1)
 {
/* Enter critical section */
   pthread_mutex_lock(&mutex);
/* If the buffer is empty, wait for new data */
   while(readIdx == writeIdx)
   {
     pthread_cond_wait(&dataAvailable, &mutex);
   }
/* At this point data are available
  Get the item from the buffer */
   item = buffer[readIdx];
   readIdx = (readIdx + 1)%BUFFER_SIZE;
/* Signal availability of room in the buffer */
   pthread_cond_signal(&roomAvailable);
/* Exit critical section */
   pthread_mutex_unlock(&mutex);

/* Consume the item and take actions (e.g. return)*/
  // ...
 }
 return NULL;
}
```

```c
int main(int argc, char *args[])
{
 pthread_t threads[MAX_THREADS];
 int nConsumers;
 int i;
/* The number of consumer is passed as argument */
 if(argc != 2) {
   printf("Usage: prod_cons <numConsumers>\n");
   exit(0);
 }
 sscanf(args[1], "%d", &nConsumers);

/* Initialize mutex and condition variables */
 pthread_mutex_init(&mutex, NULL);
 pthread_cond_init(&dataAvailable, NULL);
 pthread_cond_init(&roomAvailable, NULL);

/* Create producer thread */
 pthread_create(&threads[0], NULL, producer, NULL);
/* Create consumer threads */
 for(i = 0; i < nConsumers; i++)
   pthread_create(&threads[i+1], NULL, consumer, NULL);

/* Wait termination of all threads */
 for(i = 0; i < nConsumers + 1; i++) {
   pthread_join(threads[i], NULL);
 }
 return 0;
}
```