Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Multiple Processes
- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

Concurrency
Concurrency arises in:
- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Roadmap
- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem
Key Terms

Table 5.1 Some Key Terms Related to Concurrency

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic operation</td>
<td>A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.</td>
</tr>
<tr>
<td>critical section</td>
<td>A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.</td>
</tr>
<tr>
<td>deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.</td>
</tr>
<tr>
<td>livelock</td>
<td>A situation in which two or more processes continuously change their states in response to changes in the other processes without doing any useful work.</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.</td>
</tr>
<tr>
<td>race condition</td>
<td>A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.</td>
</tr>
<tr>
<td>starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.</td>
</tr>
</tbody>
</table>

Interleaving and Overlapping Processes

- Earlier (Ch2) we saw that processes may be interleaved on uniprocessors
- And not only interleaved but overlapped on multi-processors

Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.
A Simple Example

```c
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

A Simple Example: On a Multiprocessor

```
Process P1
    .
    chin = getchar();
    .
    chout = chin;
    .
    putchar(chout);

Process P2
    .
    chin = getchar();
    .
    chout = chin;
    .
    putchar(chout);
```

Enforce Single Access
- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

Race Condition
- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.
Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

<table>
<thead>
<tr>
<th>Degree of Awareness</th>
<th>Relationship</th>
<th>Influence That One Process Has on the Other</th>
<th>Potential Control Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes unaware of each other</td>
<td>Competition</td>
<td>Results of one process independent of the action of others</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing of process may be affected</td>
<td>Deadlock (renovable resource)</td>
</tr>
<tr>
<td>Processes indirectly aware of each</td>
<td>Cooperation by sharing</td>
<td>Results of one process may depend on information obtained from</td>
<td>Starvation</td>
</tr>
<tr>
<td>other (e.g., shared object)</td>
<td></td>
<td>others</td>
<td>Data coherence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing of process may be affected</td>
<td></td>
</tr>
<tr>
<td>Processes directly aware of each</td>
<td>Cooperation by</td>
<td>Results of one process may depend on information obtained from</td>
<td>Deadlock (consumable resource)</td>
</tr>
<tr>
<td>other (have communication primitives</td>
<td>communication)</td>
<td>others</td>
<td>Starvation</td>
</tr>
<tr>
<td>available to them)</td>
<td></td>
<td>Timing of process may be affected</td>
<td></td>
</tr>
</tbody>
</table>

Competition among Processes for Resources

Three main control problems:
- Need for Mutual Exclusion
  - Critical sections
- Deadlock
- Starvation

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Roadmap

- Principals of Concurrency
  - Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```pseudocode
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```
Special Machine Instructions

- Compare&Swap Instruction
  – also called a “compare and exchange instruction”
- Exchange Instruction

```c
int compare_and_swap (int *word, int testval, int newval) {
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Mutual Exclusion (fig 5.2)

```c
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i) {
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1) /* do nothing */;
        /* critical section */
        bolt = 0;
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... , P(n));
}
```

Exchange instruction

```c
void exchange (int register, int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```
Exchange Instruction (fig 5.2)

```c
/* program mutual_exclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi = i;
    while (true) {
        do_exchange (keyi, bolt);
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

Hardware Mutual Exclusion: Advantages
- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

Hardware Mutual Exclusion: Disadvantages
- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Some process could indefinitely be denied access.
- Deadlock is possible

Roadmap
- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- **Semaphores**
  - Monitors
  - Message Passing
  - Readers/Writers Problem
Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (semWait)
  - increment (semSignal)

Semaphore Primitives

```c
struct semaphore {
  int count;
  queueType queue;
};
void semWait(semaphore s)
{  
s.count--;
    if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process */;
    }
}
void semSignal(semaphore s)
{   
s.count++;
    if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

```c
struct binary_semaphore {
  enum {zero, one} value;
  queueType queue;
};
void semWait(binary_semaphore s)
{  
    if (s.value == one)
      s.value = zero;
    else {
      /* place this process in s.queue */;
      /* block this process */;
    }
}
void semSignal(binary_semaphore s)
{  
    if (s.queue is empty())
      s.value = zero;
    else {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
    }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- **Strong Semaphores** use FIFO
- **Weak Semaphores** don’t specify the order of removal from the queue
**Example of Strong Semaphore Mechanism**

Processors A, B, C, D

1. **Processor A**
   - Block queue
   - Semaphore
   - Ready queue

2. **Processor B**
   - Block queue
   - Semaphore
   - Ready queue

3. **Processor C**
   - Block queue
   - Semaphore
   - Ready queue

4. **Processor D**
   - Block queue
   - Semaphore
   - Ready queue

Figure 5.5 Example of Semaphore Mechanism

**Example of Semaphore Mechanism**

Processors C, D, A, B

1. **Processor C**
   - Block queue
   - Semaphore
   - Ready queue

2. **Processor D**
   - Block queue
   - Semaphore
   - Ready queue

3. **Processor A**
   - Block queue
   - Semaphore
   - Ready queue

4. **Processor B**
   - Block queue
   - Semaphore
   - Ready queue

Figure 5.5 Example of Semaphore Mechanism

**Mutual Exclusion Using Semaphores**

```c
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), ..., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

**Processes Using Semaphore**

Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

- Critical region
- Normal execution
- Blocked on semaphore lock
- Normal execution can proceed in parallel
- Note that normal regions are not serialized.
Producer/Consumer Problem

- General Situation:
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time

- The Problem:
  - Ensure that the Producer can't add data into full buffer and consumer can't remove data from empty buffer

Functions

- Assume an infinite buffer $b$ with a linear array of elements

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
</table>
| while (true) {  
  /* produce item v */  
  b[in] = v;  
  in++;  
} | while (true) {  
  while (in <= out)  
  /* do nothing */;  
  w = b[out];  
  out++;  
  /* consume item w */  
} |

Buffer

```
/* program producerconsumer */
int n;
binary_semaphore a = 1, delay = 0;
void producer()
{
  while (true) {
    /* produce item v */
    semWait(a);
    produce();
    append();
    n++;
    if (n==1) semSignal(b);delay();
    semSignal(a);
  }
}
void consumer()
{
  semWait(b);
  while (true) {
    /* consume item v */
    semWait(a);
    take();
    n++;
    semSignal(b);
    consume();
    if (n==0) semWait(b);delay();
  }
}
void main()
{
  n = 0;
  parbegin {producer, consumer};
} 
```

Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem
Possible Scenario

Table 5.4 Possible Scenario for the Program of Figure 5.9

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
<th>s</th>
<th>n</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Semaphores

/* program producer-consumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(n);
        append();
        n++;
        if (n == 1) semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    int m = 0; /* a local variable */
    semWait(s);
    while (true) {
        semWait(n);
        take();
        m++;
        if (m == 0) semWait(s);
    }
}
void main()
{
    parbegin (producer, consumer);
}

Correct Solution

```c
/* program producer-consumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        n++;
        if (n == 1) semSignal(delay);
        semSignal(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWait(s); delay;
    while (true) {
        semWait(s);
        take();
        m++;
        if (m == 0) semWait(s);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Semaphores

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

Bounded Buffer

Block on: Unblock on
Producer: insert in full buffer Consumer: item inserted
Consumer: remove from empty buffer Producer: item removed

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem
Semaphores

/* program boundedBuffer */
const int sizeOfBuffer = /* buffer size */;
semaphore s = 1, m = 0, c = sizeOfBuffer;

void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(m);
    }
}

void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(m);
        semSignal(s);
        consume();
    }
}

void main()
{
    parbegin (producer, consumer);
}

Demonstration Animations

- **Producer/Consumer**
  - Illustrates the operation of a producer-consumer buffer.

- **Bounded-Buffer Problem Using Semaphores**
  - Demonstrates the bounded-buffer consumer/producer problem using semaphores.

Roadmap

- **Principals of Concurrency**
- **Mutual Exclusion: Hardware Support**
- **Semaphores**
- **Monitors**
- **Message Passing**
- **Readers/Writers Problem**
Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.

Chief characteristics

- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

Synchronization

- Synchronization achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - **Cwait(c)**: Suspend execution of the calling process on condition c
  - **Csignal(c)**: Resume execution of some process blocked after a cwait on the same condition

Structure of a Monitor
Bounded Buffer Solution Using Monitor

```c
/* program producerconsumer */
monitor boundedbuffer;
char buffer[N]; /* space for N items */
int nextin, nextout; /* buffer pointers */
int count; /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */

void append (char x)
{  
  if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
     buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++; /* one more item in buffer */
  csignal(notempty); /* resume any waiting consumer */
}

void take (char x)
{  
  if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
  nextout = (nextout + 1) % N;
  count--; /* one fewer item in buffer */
  cnotify(notfull); /* notify any waiting consumer */
}

nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

Solution Using Monitor

```c
void producer()
{  
  char x;
  while (true) {
    produce(x);
    append(x);
  }
}

void consumer()
{  
  char x;
  while (true) {
    take(x);
    consume(x);
  }
}

void main()
{  
  parbegin (producer, consumer);
}
```

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
  - Message Passing
- Readers/Writers Problem
**Process Interaction**

- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory and with distributed systems

**Message Passing**

- The actual function of message passing is normally provided in the form of a pair of primitives:
  - send (destination, message)
  - receive (source, message)

**Synchronization**

- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - Sender and receiver may or may not be blocking (waiting for message)

**Blocking send, Blocking receive**

- Both sender and receiver are blocked until message is delivered
- Known as a *rendezvous*
- Allows for tight synchronization between processes.
Non-blocking Send

• More natural for many concurrent programming tasks.
• Nonblocking send, blocking receive
  – Sender continues on
  – Receiver is blocked until the requested message arrives
• Nonblocking send, nonblocking receive
  – Neither party is required to wait

Addressing

• Sendin process need to be able to specify which process should receive the message
  – Direct addressing
  – Indirect Addressing

Direct Addressing

• Send primitive includes a specific identifier of the destination process
• Receive primitive could know ahead of time which process a message is expected
• Receive primitive could use source parameter to return a value when the receive operation has been performed

Indirect addressing

• Messages are sent to a shared data structure consisting of queues
• Queues are called *mailboxes*
• One process sends a message to the mailbox and the other process picks up the message from the mailbox
Indirect Process Communication

- One to one
- Many to one
- One to many
- Many to many

Figure 5.18 Indirect Process Communication

General Message Format

- Header
  - Message Type
  - Destination ID
  - Source ID
  - Message Length
  - Control Information

- Body
  - Message Contents

Figure 5.19 General Message Format

Mutual Exclusion Using Messages

```c
/* program mutualexclusion */
const int n = /* number of processes */;

void P(int i)
{
  message msg;
  while (true) {
    receive (box, msg);
    /* critical section */
    send (box, msg);
    /* remainder */
  }
}

void main()
{
  create mailbox (box);
  send (box, null);
  parbegin (P(1), P(2), ..., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Producer/Consumer Messages

```c
const int capacity = /* buffering capacity */;
null = /* empty message */;

int i;
void producer()
{
  message msg;
  while (true) {
    receive (mayproduce, msg);
    msg = produce();
    send (mayconsume, msg);
  }
}

void consumer()
{
  message cmgs;
  while (true) {
    receive (mayconsume, cmgs);
    consume (cmgs);
    send (mayproduce, null);
  }
}

void main()
{
  create mailbox (mayproduce);
  create mailbox (mayconsume);
  for (int i = 1; i <= capacity; i++) send (mayproduce, null);
  parbegin (producer, consumer);
}
```

Figure 5.21 Producer/Consumer Messages
Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing

Readers/Writers Problem

- A data area is shared among many processes
- Some processes only read the data area, some only write to the area

Conditions to satisfy:
1. Multiple readers may read the file at once.
2. Only one writer at a time may write
3. If a writer is writing to the file, no reader may read it.

Readers have Priority

```c
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait [x];
        semWait (wsem);
        READUNIT();
        semWait [x];
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
    void writer()
    {
        while (true) {
            semWait (wsem);
            WRITEUNIT();
            semSignal (wsem);
        }
    }
    main()
    {
        readcount = 0;
        parbegin (reader, writer);
    }
```

Writers have Priority

```c
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
```
Writers have Priority

```c
void writer ()
{
    while (true) {
        message rmsg;
        while (true) {
            rmsg = i;
            send (readrequest, rmsg); 
            receive (mbox[1], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Message Passing

```c
void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[1], rmsg);
        READUNIT();
        rmsg = i;
        send (finished, rmsg);
    }
}
void writer(int j)
{
    message rmsg;
    while (true) {
        rmsg = j;
        send (request, rmsg);
        receive (mbox[1], rmsg);
        WRITEUNIT();
        rmsg = j;
        send (finished, rmsg);
    }
}
void controller()
{
    while (true) {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (request)) {
                receive (request, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        } else if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```