Concurrency: Mutual Exclusion And Synchronization

Arnav Gupta

April 11, 2024

Contents

1 Concurrency Basics

Arises in 3 different contexts:

1. **multiple applications**: multiprogramming was invented to allow processing time to be dynamically shared among a number of applications
2. **structured applications**: applications can be programmed as a set of concurrent processes
3. **operating system structure**: same structuring advantages apply to systems programs as OSs are also a set of processes and threads

Key terms:

- **atomic operation**: action that appears to be indivisible; no intermediate state or interruption
- **critical section**: section of code that requires access to shared resources, cannot be executed by 2+ processes at once
- **deadlock**: situation in which 2+ processes cannot proceed because each is waiting for one of the others
- **livelock**: situation in which 2+ processes keep changing state without doing any work
- **mutual exclusion**: requirement that when 1 process is in a critical section, no other process can be in a critical section accessing the same resources
- **race condition**: situation in which multiple threads or processes read and write a shared data item and the final result depends on their relative timing
- **starvation**: situation in which a runnable process is overlooked indefinitely by the scheduler, can be chosen but never is

2 Mutual Exclusion: Software Approaches

Software approaches assume mutual exclusion at memory access level: simultaneous access is serialized.

2.1 Dekker's Algorithm

First attempt has a `turn` variable that has the pid for the process allowed in the critical section.

- guarantees mutual exclusion but has two drawbacks
    1. processes must strictly alternate, so pace is dictated by the slower process
    2. if one process fails, the other is permanently blocked
- done using coroutines (pass execution control back and forth)

Second attempt has a bool vector `flag` with each index corresponding to a pid, where `true` means the process is in the critical section.

- does not guarantee mutual exclusion

Third attempt sets flag before checking if other process in critical section.

- if one process fails inside the critical section, the other process is blocked
- if one process fails outside the critical section, the other process is not blocked
- can cause deadlock if both processes set flag before checking the other's flag

Fourth attempt has each process turn off flag to enter critical section after turning it on.

- can cause livelock if each process repeatedly turns its flags on and off

The correct solution is Dekker's algorithm:

- uses both bool vector `flag` and variable `turn`
- process sets flag to `true` before checking
- while checking, if it is not a process's turn, it defers to the other process which will flip its flag

```
boolean flag[2];
int turn;
void P0() {
  while (true) {
    flag[0] = true;
    while (flag[1]) {
      if (turn ==  1) {
        flag[0] = false;
        while (turn == 1);
        flag[0] = true;
      }
    }
    /* critical section */
    turn = 1;
    flag[0] = false;
    /* remainder */
  }
}
void P1() {
  while (true) {
    flag[1] = true;
    while (flag[0]) {
      if (turn ==  0) {
        flag[1] = false;
        while (turn == 0);
        flag[1] = true;
      }
    }
    /* critical section */
    turn = 0;
    flag[1] = false;
    /* remainder */
  }
}
void main() {
  flag[0] = false;
  flag[1] = false;
  turn = 1;
  parbegin(P0, P1);
}
```

2.2 Peterson's Algorithm

A simpler algorithm that uses `turn` to resolve simultaneity conflicts.

```
boolean flag[2];
int turn;
void P0() {
  while (true) {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    /* critical section */
    flag[0] = false;
    /* remainder */
  }
}
void P1() {
  while (true) {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    /* critical section */
    turn = 0;
    flag[1] = false;
    /* remainder */
  }
}
void main() {
  flag[0] = false;
  flag[1] = false;
  parbegin(P0, P1);
}
```

3 Principles of Concurrency

In a multiprocessor sysmte, it is possible not only to interleave the execution of multiple processes but also to overlap them.

Difficulties with interleaving and overlapping:

1. sharing global resources is risky (read/write race conditions)

2. difficult for OS to optimally manage resource allocation
3. difficult to locate programming errors (less reproducable)

Sharing main memory permits efficient and close interaction among processes, but can lead to problems:

- if one process updates a global variable and is interrupted, another process may alter the variable before the first process can use its value

Necessary to protect shared global variables by controlling the code that accesses the variable.

3.1 Operating System Concerns

Concerns are:

1. OS must be able to keep track of processes
2. OS must allocate and deallocate resources for each active process (processor time, memory, files, IO devices)
3. OS must protect data and physical resources of each process against unintended interference by other processes
4. functioning of a process and its output must be independent of the relative speed of execution

3.2 Process Interaction

For processes unaware of each other:

- OS must be concerned about competition for resources
- results of one process independent of the action of others
- timing of process may be affected
- potential problems are mutual exclusion, deadlock of a renewable resource, starvation

For processes indirectly aware of each other:

- exhibit cooperation by sharing
- results of one process may depend on information obtained from others
- timing of process may be affected

- potential problems are mutual exclusion, deadlock of a renewable resource, starvation, data coherence

For processes directly aware of each other:

- exhibit cooperation by communication
- results of one process may depend on information obtained from others
- timing of process may be affected
- potential problems are deadlock of a consumable resource, starvation

If competition exists:

- each process should leave the state of any resource that it uses unaffected
- since a process must wait to use a resource, it will be slowed down or may never get access to it

Problems that must be solved for competition are:

1. **Mutual exclusion**: ensuring only one program using the critical resource in the critical section at a time
2. **Deadlock**: processes exist a loop waiting on other processes
3. **Starvation**: process is indefinitely denied access to a resource, even without deadlock

Processes themselves must express the requirement for mutual exclusoin, with support from the OS to follow this.

If cooperation by sharing exists:

- processes must cooperate to ensure shared data is properly managed
- control mechanisms must ensure the integrity of shared data

If cooperation by communication exists:

- processes must synchronize various activities through messages
- deadlock and starvation can occur, through awaiting communication from other processes

3.3 Requirements for Mutual Exclusion

1. **Enforcement**: only one process at a time allowed in the critical section for a resource
2. **No Interference**: a process that halts in its noncritical section must do so without interfering with other processes
3. **No Deadlock or Starvation**
4. **No Unnecessary Delay**: a process must not be delayed access to a critical section when there is no other process using it
5. **No Assumptions**: no assumptions made about relative process speeds or number of processors
6. **Finite Time**: a process remains inside its critical section for a finite time only

Main mechanisms for Mutual Exclusion:

- software with Dekker's algorithm (slow)
- special hardware instructions
- support in the OS or a programming language

4 Mutual Exclusion: Hardware Support

4.1 Interrupt Disabling

Disabling interrupts guarantees mutual exclusion for a uniprocessor system.

Costs of interrupt disabling:

- efficiency is degraded since there is limited ability to interleave processes.
- does not work for multiprocessor architecture, since a process on another processor could invalidate mutual exclusion

4.2 Special Machine Instructions

Special instructions carry out two actions atomically during one instruction fetch cycle.

Compare&Swap instruction (carried out atomically)

- `testval` is compared to memory location
  - if equal then a swap occurs with `newval`
- returns either the old value or whether or not the swap occurred

```
const int n = 5;
int bolt;
void P(int i) {
  while (true) {
    while (compare_and_swap(bolt, 0, 1) == 1);
    /* critical section */
    bolt = 0;
    /* remainder */
  }
}
void main() {
  bolt = 0;
  parbegin(P(1), P(2), ..., P(n));
}
```

Exchange instruction (carried out atomically)

- exchanges contents of register with that of memory location

```
const int n = 5;
int bolt;
void P(int i) {
  while (true) {
    int keyi = 1;
    do (exchange(&keyi, &bolt))
         while (keyi != 0);
    /* critical section */
    bolt = 0;
    /* remainder */
  }
}
void main() {
  bolt = 0;
  parbegin(P(1), P(2), ..., P(n));
}
```

Advantages of machine-instruction approach:

- applicable to any number of processes for both uniprocessor and multiprocessor
- simple and easy to verify
- can be used to support multiple critical sections, each defined by its own variable

Disadvantages of machine-instruction approach:

- busy waiting: process waiting for access to a critical section consumes processor
- starvation possible: when a process leaves a critical section and more than one process is waiting, some process could never be chosen
- deadlock possible: higher priority processes could end up waiting on lower priority processes inside critical sections

5 Semaphores

Common concurrency mechanisms:

- **semaphore**: integer value used for signaling, with atomic operations of initialize, decrement, and increment where decrement can block a process and increment can unblock a process
- **binary semaphore**: semaphore that takes on only 0 and 1
- **mutex**: similar to binary semaphore but the process that locks a mutex must unlock it
- **condition variable**: data type used to block a process or thread until a condition is true
- **monitor**: programming language construct to encapsulate variables, critical sections, and initialization code in one data type, where it may only be accessed inside critical sections by one process
- **event flags**: memory word used as a synchronization mechanism with bits representing events that must be waited on
- **messages**: means for two processes to exchange info, possibly for synchronization
- **spinlocks**: mutual exclusion mechanism in which a process

Two or more processes can cooperate by means of simple signals.

For a semaphore `s`:

- to transmit a signal, a process executes `semSignal(s)`
- to receive a signal, a process executes `semWait(s)`, which suspends the process until a signal is transmitted

Semaphore is an integer variable with only 3 operations defined:

1. initialized to nonnegative integer value
2. `semWait` decrements the semaphore value, if it becomes negative the process executing `semWait` is blocked, otherwise it continues execution
3. `semSignal` increments the semaphore value, if it is less than or equal to zero, a process blocked by `semWait` is unblocked

Consequences of semaphore definition:

1. no way to know before `semWait` whether it will block
2. after a process increments a semaphore and another process wakes up, both processes run concurrently
3. when calling `semSignal`, the number of unblocked processes may be 0 or 1

Operations on binary semaphore are:

1. initialized to 0 or 1
2. `semWaitB` checks the semaphore value, if 0 then the process executing `semWaitB` is blocked, if 1, then the value is changed to 0 and the process continues
3. `semSignalB` checks if value is 0 (processes blocked), if so then one blocked process is unblocked, if no processes blocked the value is set to 1

Mutex is a programming flag to grab and release an object:

- mutex locks (set to 0) when non-shareable data acquired or processing started that cannot be performed elsewhere
- mutex unlocks when the data is no longer needed or the routine is finished

Semaphores use a queue to hold processes waiting on the semaphore:

- strong semaphores use FIFO for release, guarantee freedom from starvation
- weak semaphores do not specify the order of release, no guarantee of freedom from starvation

5.1 Producer/Consumer Problem

General statement of problem:

- one or more producers generating some type of data and placing into buffer
- single consumer takes items out of buffer one at a time
- system is to be constrained to prevent the overlap of buffer operations (only one agent accessing the buffer at a time)

Order of semaphore operations can sometimes matter, especially for `semWait` where there is potential for deadlock.

Infinite-buffer producer/consumer problem with binary semaphores

```
int n;
binary_semaphore s = 1, delay = 0;
void producer() {
  while (true) {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n == 1) semSignalB(delay);
    semSignalB(s);
  }
}
void consumer() {
  int m;
  semWaitB(delay);
  while (true) {
    semWaitB(s);
    take();
    n--;
```

```
    m = n;
    semSignalB(s);
    consume();
    if (m == 0) semWaitB(delay);
  }
}
void main() {
  n = 0;
  parbegin(producer, consumer);
}
```

Infinite-buffer producer/consumer problem with semaphores

```
semaphore n = 0, s = 1;
void producer() {
  while (true) {
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}
void consumer() {
  while (true) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    consume();
  }
}
void main() {
  parbegin(producer, consumer);
}
```

For a finite buffer, it can be treated as circular with pointer values expressed module the buffer size. Block occurs on:

- insert in full buffer for producer
- remove from empty buffer for consumer

Unblock occurs on:

- item inserted for consumer
- item removed for producer

Finite-buffer producer/consumer problem with semaphores

```
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer() {
  while (true) {
    produce();
    semWait(e);
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}
void consumer() {
  while (true) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    semSignal(e);
    consume();
  }
}
void main() {
  parbegin(producer, consumer);
}
```

5.2 Implementation of Semaphores

Semaphores must be implemented as atomic primitives; only one process at a time may manipulate a semaphore with either `semWait` or `semSignal`.

Can be done with:

- hardware or firmware

- software schemes (Dekker's, Peterson's)
- hardware-supported schemes for mutual exclusion
  - can be done with `compare_and_swap` with a flag or by inhibiting interrupts

6 Monitors

May be difficult to produce a correct program using semaphores.

6.1 Monitor with Signal

Monitor: a software module consisting of one or more procedures, an initialization sequence, and local data

Chief chracteristics are:

1. local data variables are only accessible only by the monitor's procedures and not by any external procedure
2. a process enters the monitor by invoking one of its procedures
3. only one process may be executing in the monitor at a time, any other processes that have invoked the monitor are blocked, waiting for the monitor to become available

Monitors support synchronization using conditional variables contained within the monitor and accessible only within the monitor.

Condvars operated on by two functions:

- `cwait(c)`: suspend execution of the calling process on condition `c`, monitor is now available for use by another process
- `csignal(c)`: resume execution of some process blocked after a `cwait` on the same condition:
  - if there are multiple such processes, choose one
  - if none, do nothing

Different from semaphores as any signals that tasks aren't waiting on are lost.

```
monitor boundedbuffer;
char buffer[N];
```

```
int nextin, nextout;
int count;
cond notfull, notempty;

void append(char x) {
  if (count == N) cwait(notfull);
  buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++;
  csignal(notempty);
}

void take(char x) {
  if (count == 0) cwait(notempty);
  x = buffer[nextout];
  nextout = (nextout + 1) % N;
  count--;
  csignal(notfull);
}

{
  nextin = 0; nextout = 0; count = 0;
}
void producer() {
  char x;
  while (true) {
    produce();
    append();
  }
}
void consumer() {
  char x;
  while (true) {
    take();
    consume();
  }
}
void main() {
  parbegin(producer, consumer);
}
```

6.2 Alternate Models of Monitors with Notify and Broadcast

Currently, a process issuing the `csignal` must either immediately exit the monitor or be blocked on the monitor, which has 2 drawbacks:

1. if the process issuing the `csignal` has not finished with the monitor, two additional process switches are required
2. process scheduling associated with a signal must be reliable, no other process can interrupt this switching

Bounded-buffer Monitor Code for Mesa Monitor

```
void append(char x) {
  while (count == N) cwait(notfull);
  buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++;
  cnotify(notempty);
}

void take(char x) {
  while (count == 0) cwait(notempty);
  x = buffer[nextout];
  nextout = (nextout + 1) % N;
  count--;
  cnotify(notfull);
}
```

`cnotify` differs in that the condition queue is notified, but the signalling process continues to execute. `cbroadcast` can also cause all processes waiting on a condition to be placed into a ready state.

Mesa monitor less error prone and more modular than usual Hoare monitor.

7 Message Passing

When processes interact, must ensure synchronization and communication. Can be done with message passing, which has 2 primitives:

- `send(destination, message);`
- `receive(source, message);`

Sender and receiver can be blocking or nonblocking, of which there are 3 common combinations

1. **blocking send, blocking receive**: both sender and receiver blocked until message is delivered, allows for tight synchronization
2. **nonblocking send, blocking receive**: sender may continue but receiver is blocked until message arrives
3. **nonblocking send, nonblocking receive**: neither party must wait

Nonblocking send is most natural since a sender can then send multiple different messages. Blocking receive is most natural since receivers generally expect information before proceeding.

7.1 Addressing

With direct addressing

- the `send` primitive includes a specific identifier

of the destination process.

- the `receive` primitive either:
  - requires that the process explicitly designate a sending process
  - uses implicit addressing: the source possesses a value returned when the receive has bene performed

With indirect addressing

- messages are sent to queue that holds messages, called mailbox
- decoupling sender and receiver allows for more flexibility in use of messages
- relationship between senders and receivers can be:
  - one-to-one: private communication between two processes
  - many-to-one: client-server interaction, creates port
  - one-to-many: broadcasting to set of processes
  - many-to-many: multiple server processes to provide concurrent service to multiple clients
- mailboxes owned by creating process (ex. port)

7.2 Message Format

Message typically divided into header and body. Header contains:

- message type
- destination ID
- source ID
- message length
- control information

Body contains message contents.

Messages can be queued based on priority.

7.3 Mutual Exclusion

Mutual Exclusion using Messages

```
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));
}
```

Bounded-buffer Producer/Consumer using Messages

```
const int capacity = /* number of processes */;
int i;
void producer() {
  message pmsg;
  while (true) {
```

```
    receive(mayproduce, pmsg);
    pmsg = produce();
    send(mayconsume, pmsg);
  }
}
void consumer() {
  message cmsg;
  while (true) {
    receive(mayconsume, cmsg);
    consume();
    send(mayproduce, null);
  }
}
void main() {
  create_mailbox(mayproduce);
  create_mailbox(mayconsume);
  for (int i = 1; i <= capacity; i++) send(mayproduce, null);
  parbegin(producer, consumer);
}
```

8 Readers/Writers Problem

Problem defined as follows: there is a data area shared among processes, some of which only read and some of which only write. The conditions on these are:

1. any number of readers may simultaneously read the file
2. only one writer at a time may write to the file
3. if a writer is writing to the file, no reader may read it

8.1 Readers Have Priority

```
int readcount;
semaphore x = 1, wsem = 1;

void reader() {
  while (true) {
    semWait(x);
    readcount++;
```

```
    if (readcount == 1) {
      semWait(wsem);
    }
    semSignal(x);
    READUNIT();
    semWait(x);
    readcount--;
    if (readcount == 0) {
      semSignal(wsem);
    }
    semSignal(x);
  }
}

void writer() {
  while (true) {
    semWait(wsem);
    WRITEUNIT();
    semSignal(wsem);
  }
}

void main() {
  readcount = 0;
  parbegin(reader, writer);
}
```

When one writer is writing, no other writer or reader can access the shared data.

The first reader waits on `wsem` (in case of writing), subsequent readers need not wait since first one guarantees no writing.

Last reader finishing up signals `wsem` that writing can occur as no other readers accessing data.

8.2 Writers Have Priority

Prevents writer starvation by guaranteeing no new readers once at least one writer desires to write.

```
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);
  }
}

void writer() {
  while (true) {
    semWait(y);
    writecount++;
    if (writecount == 1) {
        semWait(rsem);
    }
    semSignal(y);
    semWait(wsem);
    WRITEUNIT();
    semSignal(wsem);
    semWait(y);
    writecount--;
    if (writecount == 0) {
        semSignal(rsem);
    }
```

```
    semSignal(y);
  }
}

void main() {
  readcount = 0;
  parbegin(reader, writer);
}
```