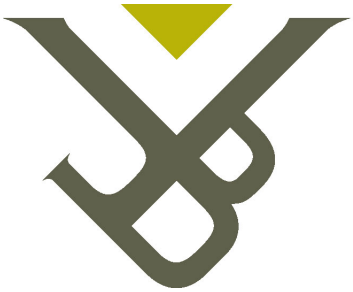

Vrije Universiteit Brussel

Operating Systems and Security Project Report: Synchronization and Communication

Mathijs Saey (94451)
mathsaey@vub.ac.be

2nd Master of Science in Applied Sciences and Engineering: Computer Science

January 15, 2014

Abstract

Threads and processes are the bread and butter of a modern day operating system. To facilitate dealing with threads and processes, most operating systems have introduced useful constructs to allow processes to work in a shared environment.
In this paper, we present an overview of the available techniques that facilitate inter-thread and inter-process communication and synchronization on multiple operating systems, namely Microsoft Windows and Mac OS X. To show the use of these techniques, we also introduce a few sample programs showcasing the use of inter-process communication.

Contents

1 Introduction

Computers have been able to run multiple programs since the early 1960's due to the introduction of concepts such as multiprogramming and time sharing. Even though an effective scheduler is all that is strictly required to run multiple programs on the same machine, modern day operating systems also need to deal with the need to make these processes communicate with one another. Besides that, these processes can also try to modify the same resources, leading to disaster if this resource is not accessed correctly.

Fortunately, modern-day operating systems provide us with a sufficient amount of tools to handle most of these situations without reinventing the wheel every time we need to do so. These tools come in a wide amount of flavors, to cover multiple scenarios. However, finding the right tool in this box can still be hard for a novice to the field of multi-process programming.

This paper provides an overview of some of the most common of these tools, along with some examples of their use in 2 modern day operating systems, Microsoft Windows, and Mac OSX.

The attentive reader might have noticed that we did not talk about threads up to this point. However, processes and threads are 2 very similar constructs, a process and a thread are both independent sequences of execution. The main difference is that threads tend to run in the same address space while processes don't. In this paper we will show instances of both process and thread communication and provide code examples for both thread and process based communication. Most of the showcased techniques work for both.

About the code

Before we go into detail about the various tools a few practical remarks about the code might be in place.

We decided to use the python [4] programming language for our code examples. We did this because python runs on may different operating systems without too much hassle. Python also serves as an abstraction layer for the underlying system which means that we can focus on the underlying idea without worrying about it's implementation on the specific operating system.

We should also explain python's *with* block statement. Simply put, with receives a lock and code block as input, with then automatically calls *lock.acquire()* before starting the block. After finishing the block, with calls *lock.release()*.

Thus, the following pieces of code are equivalent:

```
with lock:
    statement()
```

```
lock.acquire()
statement()
lock.release()
```

Throughout the code, we use *with* a lot, we only use a lock explicitly in the code that showcases mutexes.

It's important to note that running the provided python code on windows should be done from the command prompt. This is done with the following command:

```
> python progname.py
```

This is required since not all of the multiprocessing module's functionality is available in the interactive interpreter [3]. All of the listed code was tested on python 3.3.3 on Windows 7 and OSX 10.9.1.

Finally, in case of any problems, a copy of the source code, report and related files can be found at `https://github.com/mathsaey/OSSECproj`.

2 Mechanisms

In this section, we introduce the various mechanisms at our disposal, each mechanism is accompanied by a small explanation. A program demonstrating a possible use case for each of these scenarios is presented in the next section.

We differentiate between 2 different categories of mechanisms, synchronization and communication.

The first mechanism category, synchronization, allows threads or processes to share a common resource. The threads or processes do not need to be aware of the other threads/processes, they simply need to access the shared resource in a safe way.

The second mechanism, communication, allows thread or processes to "talk" to each other, this can be used to send some requests or to pass some data. The processes need to be aware of the other other processes, but they don't need to be aware of the exact nature of the other processes.

In practice, both of these mechanisms are utilized alongside one another. For instance, using a file to communicate between different processes often requires synchronizing the access to this file.

2.1 Synchronization

2.1.1 Locks and Mutexes

The first and perhaps most well known way of synchronizing access to a shared resource is a simple, straightforward lock. When a thread reaches it's *critical section*, the part of its code where it needs access to a shared resource, it simply requests a lock on the shared resource. If the resource is currently not locked, the thread acquires the lock, performs its critical section and opens the lock. Any other thread that attempts to obtain the lock will block until the lock is available.

A mutex is simply a lock that is not limited to the current process. Instead, a mutex is managed by the system.

OS Task 5 [5] showcases using mutexes in the POSIX thread model.

2.1.2 Semaphores

Semaphores allow us to limit access to a certain resource. A semaphore allows us to specify the amount of threads that are allowed to access a resource. For instance, we could use a semaphore to dictate that only 10 processes can access a given database at the same time to avoid overloading our database with requests.

A mutex can be seen as a more limited semaphore. Indeed, a *binary semaphore*, a semaphore that only allows one thread to access our resource at any given time, is functionally equivalent to a mutex.

2.1.3 Condition Variables

A Condition Variable on it's own does not allow us to safely access a shared resource, however condition variables can be used in combination with locks to create monitors, discussed in the next section.

A condition variable can be seen as a collection of threads that are waiting for a certain condition to be met. Other threads can use this variable to notify a thread that a condition has been met. The main reason for doing this is to avoid *busy waiting* threads, threads that wake up, check a condition and go back to sleep. Using a condition variable removes this overhead at the expense of other threads notifying the condition variable at the right moment.

2.1.4 Monitors

As we mentioned in the previous section, condition variables are not used on their own to safely access a resource. Instead, we use a combination of locks and condition variables access our shared resource. A combination of a condition variable and a lock is called a monitor. It's worth noting that some thread models (such as the posix thread model [1]) don't have a concept known as monitors, in these models, the programmer is tasked with creating a lock himself.
A condition variable is always used in conjunction with a mutex lock [1, 8.1. Condition Variables Overview]

The condition variable allows a thread in it's critical section to give up its lock and wait for a condition to be met. Another thread can now signal this thread about this condition through the condition variable. The first thread will now continue it's execution once it re-obtains the original lock. Typically, monitors are used to synchronize many threads, and a thread can choose to notify one or more of the waiting threads.

2.2 Communication

2.2.1 (Memory Mapped) Files

Almost anybody that has ever used a computer knows about files. While files generally convey thoughts about long term data storage, files can also easily be used to exchange information between different processes. One process can simply write something to the file, another program can read this data, and perform some action depending on the exact contents of the file. Modifying files may seem like a rather primitive way to achieve communication, but they are a simple way to achieve this, since files exist on almost any current-day operating system. This last characteristic also makes using files a strategy that is very portable between many operating systems.

When using this method, programmers should take care to synchronize access to the file.

A memory mapped file is a file that is present on the random access memory, making it faster to read and write to this file.

2.2.2 Signals

Signals are notifications that can be sent to processes or threads, it's important to know that signals are sent asynchronously. An example of using a signal would be pressing `ctrl + c` to interrupt a process running in a terminal. Doing so sends the running process a `SIGINT` signal (on most UNIX systems), requesting it to interrupt execution. A program can catch this signal and respond accordingly. Another use of signals is the kernel notifying a running process about a segmentation fault. Signals do not carry any extra data. Another example of signal communication can be found when using monitors, waking up threads that were waiting for a condition to be met generally happens by sending a signal to the waiting thread.

In short, we can say that signals are a lightweight way of notifying processes about predefined events.

2.2.3 Sockets

A socket is the endpoint of a network connection, a process can ask the operating system to create a socket on a given address and "listen" to it. A process that listens to a socket accepts the data stream that is sent through the socket, and can respond to data sent over over the network. The process on the other end of this connection can be connected to the socket over a network, but it can also be a process running on the same machine.

For instance, in SEC task 3 [7] we used a program called *FakeSMTP* to locally accept SMTP connections. This program simply listened on a given local port and accepted all traffic that was sent to this port. Sending an email to FakeSMTP from my mail client is one example of inter-process communication through (local) sockets.

Sockets are another popular communication technique, since they are present on most operating systems.

2.2.4 Message Queues

Messages queues can be seen as a "mailbox" for a given process or thread. The idea behind message queues is quite simple, a process defines a queue which can accept external messages in an asynchronous manner. Other processes can now add data, known as messages, to the queue. The receiving thread can now read the contents of the queue and perform some action based on the contents of the queue.

For instance, AmbientTalk [2], a distributed programming language developed at the VUB, uses message queues to send asynchronous messages to *remote* objects, objects that are located in a different actor (typically on another device). This mechanism allows the calling thread to send a message to a different thread without blocking. Message Queues were also showcased in OS task 6 [6].

2.2.5 (Named) Pipes

Pipes are perhaps best known from the UNIX shell. Using pipes, a user can chain together different processes, passing the output from one processes as input to the next one. Pipes communicate by sending data over the standard input and output stream.

However, pipes are not limited to being created on a shell, a process can also programmatically create read and write pipes and pass them along to other processes.

Named pipes work in the same way as standard pipes with one notable exception, instead of receiving data through standard input and sending data through standard output, named pipes use a file to communicate. A process can now access this file by name and either read the pipe or write to it.

The named pipe model seems to be very similar to the file mechanism discussed in section 2.2.1. However named pipes are implemented differently depending on the underlying OS, for instance named pipes on windows are automatically deleted once the last reference to this file is closed. On Unix systems, named pipes can also be created without actually writing the file to the hard drive. Furthermore, named pipes offer a level of abstraction that files do not offer.

2.2.6 Shared Memory

Shared memory is exactly what it sounds like, a process shares memory with another process, which allows us to share global variables, locks, and any other data that the process wishes to expose. This has the benefit of being far faster than most of the other mechanisms. Furthermore, this also allows us to eliminate duplicate data that would otherwise exist in both processes. It's also worth noting that threads running as a part of the same process are running on shared memory by default.

Care should be taken to synchronize access to shared memory to avoid multi-

ple processes modifying the same data at the same time. We refer the reader to the techniques discussed in section 2.1 in order to do so.

3 The Programs

In this section, we introduce the various example programs that we wrote to showcase the various communication and synchronization techniques.

3.1 Locks and Mutexes

We decided to showcase mutexes by having different processes accessing a shared file. The idea behind is the example is the simulation of multiple processes writing to a shared log file. The mutex prevents the processes from accessing the file at the same time.

```
1 from multiprocessing import Process, Lock
2
3 def writeMessage(idx, message, lock):
4   message = str(idx) + " mu: " + message + '\n'
5   lock.acquire()
6   f = open('mutexfile.txt', 'a')
7   f.write(message)
8   lock.release()
9
10 if __name__ == '__main__':
11   processes = []
12   lock = Lock()
13
14   for idx in range(0, 10):
15     p = Process(
16       target = writeMessage,
17       args = (idx, "writing to shared resource!", lock))
18     p.start()
19     processes += [p]
20   for p in processes:
21     p.join()
```

Listing 1: Using locks to manage access to a file.

3.2 Semaphores

In our code example, semaphores are showcased by having a few threads access a "resource" that can only handle a few requests at the same time. We simulate this resource by forcing the process to sleep for a randomly determined amount of seconds. Only 3 processes are allowed to access our resource at the same time. Using semaphores, we can ensure that only 3 threads are accessing our resource at any given time.

```
1 from threading import Semaphore, Thread
2 from time import sleep
3 import random
4
5 def access(idx, sem):
6   with sem:
7     print(idx, "starting critical section")
8     sleep(random.randint(0,3))
9     print(idx, "finished critical section")
10
11 if __name__ == '__main__':
12   random.seed()
13   threads = []
14   sem = Semaphore(3)
15
16   for idx in range(0, 10):
17     t = Thread(
18       target = access,
19       args = (idx, sem))
20     t.start()
21     threads += [t]
22   for t in threads:
23     t.join()
```

Listing 2: Using semaphores to limit access.

3.3 Condition Variables and Monitors

We showcase condition variables by creating their typical usage scenario, we have a thread that will only do something useful if a certain condition is met. More specifically, we create 2 threads, a producer and a consumer thread. The producer produces 100 random integers and adds them to a shared resource. The consumer's only job is to display an alert if a lucky number (22) appears in the resource. In order to do this, the consumer decides to wait on the condition variable. If the producer creates our lucky number, it notifies the condition variable, which notifies our consumer thread.

In python, a condition variable automatically creates a matching lock. Thus a python condition variable can be seen and treated as a monitor.

```
1 from threading import Thread, Condition
2 import random
3
4 resource = []
5 condVar = Condition()
6
7 def produce():
8   global resource
9   for ctr in range(0,100):
10    with condVar:
11      i = random.randint(0, 100)
12      if i is 22: condVar.notify()
13      resource += [i]
14
15 def consume():
16   with condVar:
17     while resource.count(22) is 0:
18       condVar.wait()
19     print("Lucky number 22 found!", resource)
20
21 if __name__ == '__main__':
22   random.seed()
23
24   producer = Thread(target = produce)
25   consumer = Thread(target = consume)
26   consumer.daemon = True
27   producer.start()
28   consumer.start()
29   producer.join()
```

Listing 3: Using condition variables to get notified when a condition occurs.

3.4 (Memory Mapped) Files

We demonstrate the use of files by having a few processes attempt to discover each other. In order to do this, each and every one of these programs writes its name to a shared file. After adding their name, the processes scan the file for the names of other processes.

```
from multiprocessing import Process, Lock

path = 'names.txt'

def leaveName(name, lock):
  name = str(name) + '\n'
  with lock:
    f = open(path, 'a')
    f.write(name)

def getNames(lock):
  with lock:
    f = open(path, 'r')
    str = f.read()
  return str.strip().split('\n')

def discover(name, members, lock):
  leaveName(name, lock)
  print(name, "left name")
  names = []
  found = 0
  while found is not members:
    names = getNames(lock)
    found = len(names)
  print(name, "found:", names)

if __name__ == '__main__':
  open(path, 'w')
  processes = []
  lock = Lock()
  members = 5

  for idx in range(0, members):
    p = Process(
      target = discover,
      args = (idx, members, lock))
    p.start()
    processes += [p]
  for p in processes:
    p.join()
```

Listing 4: Using a file to communicate.

3.5 Signals

In order to showcase signals, we created 2 processes. The first process performs a task that is time critical, if the task is not finished in time, the task should not finish at all. In order to enforce this, we create a second process called the guard process.

The guard process's only goal is ensuring that the process finishes in time or not at all. When necessary, the guard process sends a kill signal to the other process.

```
1 from multiprocessing import Process
2 from time import sleep
3 import signal
4 import os
5
6 # The kill signal is different depending
7 # on the platform we are running on
8 def getKillSig():
9   try:
10    return signal.SIGKILL
11  except AttributeError:
12    return signal.CTRL_C_EVENT
13
14 def guardProcess(process, timeout):
15   sleep(timeout)
16   if process.is_alive():
17     pid = process.pid
18     os.kill(pid, getKillSig())
19     print("Process killed...")
20     os._exit(1)
21   else:
22     print("Process finished...")
23     os._exit(0)
24
25 def longProcess():
26   sleep(60)
27
28 if __name__ == '__main__':
29   process = Process(target = longProcess)
30   process.start()
31   guardProcess(process, 2)
```

Listing 5: Sending a signal to terminate a process

3.6 Sockets

We showed the use of sockets by creating a simple local ping server. The server listens to a port on localhost. A client can connect to the server and send it some data, the server will echo this data and answer the client by sending it a "pong" message.

```
1 from multiprocessing import Process
2 import socketserver
3 import socket
4
5 BUFFER_SIZE = 2048
6 HOST = "localhost"
7 PORT = 1234
8
9 class requestHandler(socketserver.BaseRequestHandler):
10   def handle(self):
11     message = self.request.recv(BUFFER_SIZE)
12     message = message.decode("utf-8")
13     address = self.client_address[0]
14     port = self.client_address[1]
15     print("Server received:", message, "from", address, ":", port)
16     self.request.sendall(bytes("pong","utf-8"))
17
18   def finish(self):
19     self.server.shutdown()
20
21 def runServer():
22   server = socketserver.TCPServer((HOST, PORT), requestHandler)
23   server.serve_forever()
24
25 def runClient():
26   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
27   sock.connect((HOST, PORT))
28   sock.sendall(bytes("ping","utf-8"))
29   reply = sock.recv(BUFFER_SIZE)
30   reply = reply.decode("utf-8")
31   print("Client received:", reply)
32
33 if __name__ == '__main__':
34   server = Process(target = runServer)
35   client = Process(target = runClient)
36   server.daemon = True
37   server.start()
38   client.start()
39   client.join()
```

Listing 6: Using sockets for low level communication

3.7 Message Queues

In order to show a possible use of message queues, we followed the example of AmbientTalk and used the message queue as a way of asking a different process to perform some task. Our program defines a worker process that only exists to process requests from other processes. In order to send a process to the worker thread, one simply adds a function and its arguments to the inQueue of the worker process. The worker process can now remove this data from the queue and execute this function with the given arguments. The results of function execution are added to the outQueue of the worker process, which can be read by other processes.

```
1 from multiprocessing import Process, Queue
2
3 def workProcess(inQueue, outQueue):
4   while True:
5     tuple = inQueue.get()
6     func = tuple[0]
7     args = tuple[1]
8     res = func(*args)
9     outQueue.put(res)
10
11 def sendFunction(function, args, inQueue, outQueue):
12   inQueue.put((function, args)),
13   return outQueue.get()
14
15 def aFunction(x,y): return x + y
16
17 if __name__ == '__main__':
18   inQueue = Queue()
19   outQueue = Queue()
20
21   worker = Process(
22     target = workProcess,
23     args = (inQueue, outQueue))
24   worker.daemon = True
25   worker.start()
26
27   r = sendFunction(
28     aFunction,
29     (3,4),
30     inQueue,
31     outQueue)
32
33   print(r)
```

Listing 7: Using a queue for message dispatching

3.8 (Named) Pipes

We show the use of pipes by creating our own version of the *pipes and filters* design pattern. In this design pattern, each filter is a module that performs one specific function. These filters are connected by pipes.

In our pipes and filters implementation, every filter runs on a separate process. We uses pipes to connect these different processes. In order to implement this, we created 2 functions. The first, *createFilter* takes a function and 2 pipe-ends. One of these ends is used to get incoming data, while the other is used to send outgoing data. These ends are not allowed to be connected to the same pipe. The second function, *createProcess*, receives a list of functions. It creates a filter for each of these processes (with the createFilter function) and starts a process that executes this filter. It also ensures that the filters are properly connected through pipes.

We used this machinery to create a simple pipeline that receives a sentence, which it converts to camel case. It does this in 4 steps, each realized in a specific filter.

- The first filter splits the sentence into a list of words.
- The second filter capitalizes the first letter of every word.
- The third filter merges these words with no spaces in between.
- The final filter changes the first letter of our string to lowercase.

```
from multiprocessing import Process, Pipe

def createFilter(func, inc, out):
  while True:
    arg = inc.recv()
    res = func(arg)
    out.send(res)

# pIn =pipe= pOut FUNC toNext =pipe= pEnd
def createProcesses(functions):
  pIn, pOut = Pipe()
  pipeLineIn = pIn

  for f in functions:
    (toNext, pEnd) = Pipe()
    p = Process(
        target = createFilter,
        args = (f, pOut, toNext))
    p.daemon = True
    p.start()
    pOut = pEnd
  return(pipeLineIn, pEnd)

def split(str): return str.split(' ')

def capitalize(lst): return list(map(lambda s: s.capitalize(), lst))

def merge(lst):
  res = ""
  for s in lst: res += s
  return res

def decapitalizeFirst(str):
  head = str[0]
  tail = str[1:]
  return head.lower() + tail

if __name__ == '__main__':
  lst = [split, capitalize, merge, decapitalizeFirst]
  t = createProcesses(lst)
  begin = t[0]
  end = t[1]

  begin.send("Make this sentence camelCase")
  res = end.recv()
  print(res)
```

Listing 8: Using pipes to pass data.

3.9 Shared Memory

In our final example, we use shared memory to frequently check the status of another process. This can be useful if we want to display the state of another process that is continually generating new data. An example would be a daemon that is monitoring system performance, while another process displays this data to the user.

Our process continually replaces data in the shared memory, another process, called the monitor, frequently checks the state of this memory and displays its contents on the screen. Finally, another piece of shared data is used to signal the processes to stop after an arbitrary amount of time.

```
1 from multiprocessing import Process, Value, Array
2 from time import sleep
3 import random
4
5 def runProcess(arr, remainActive):
6   while remainActive.value:
7     for idx in range(len(arr)):
8       arr[idx] = random.randint(0,100)
9     sleep(1)
10
11 def runMonitor(arr, remainActive):
12   while remainActive.value:
13     print(arr[:])
14     sleep(2)
15
16 if __name__ == '__main__':
17   random.seed()
18   arr = Array('i', range(5))
19   remainActive = Value('i', 1)
20
21   process = Process(target = runProcess, args = (arr, remainActive))
22   monitor = Process(target = runMonitor, args = (arr, remainActive))
23   process.daemon = True
24   monitor.daemon = True
25   process.start()
26   monitor.start()
27
28   sleep(10)
29   remainActive.value = 0
30
31   process.join()
32   monitor.join()
```

Listing 9: Sharing memory between processes

4 Conclusion

In this paper, we discussed 10 common instruments to facilitate intra-procedural communication and synchronization. Furthermore, we shared a few examples of using these instruments for various ends.

While reasoning about possible use cases for these tools, it became clear to us that most of these tools have their own specific niche, their very own problem that they tackle. It is up to the programmer to find the right tool for the right job, but this is only possible if the operating system offers the right tools. Therefore, it is only reasonable that most of these tools find a way to reappear, irregardless of the system the programmer is working on.

5 References

[1] Blaise Barney. *POSIX Threads Programming*. URL: `https://computing.llnl.gov/tutorials/pthreads/`.

[2] Tom Van Cutsem. *What is AmbientTalk about?* URL: `http://soft.vub.ac.be/amop/`.

[3] *Multiprocessing in Python*. URL: `http://docs.python.org/3/library/multiprocessing.html`.

[4] *Python Programming Language Official Website*. URL: `http://www.python.org/`.

[5] Mathijs Saey. "OSSEC Task Report: OS Task 5: Mutual exclusion". OSSEC Task report. 2013.

[6] Mathijs Saey. "OSSEC Task Report: OS Task 6: Communication between threads". OSSEC Task report. 2014.

[7] Mathijs Saey. "OSSEC Task Report: SEC Task 3: Network Sniffing". OSSEC Task report. 2013.