IPC, Synchronisation and Threads

1 Interprocess Communication (IPC)

1.1 Why Do Processes Intercommunicate?

- If they never did, then all input to a process would have to come from the user! There are several reasons why IPC must be provided. Often a problem is broken into several stages, each handled by a **process** (e.g compilation), so each process passes information to the next stage.
- Sometimes a package is broken up into several parts (e.g for an accounting package: inventory, credits, debits, invoicing, payroll). Each part will need to pass/obtain information to/from another part (e.g sales affect inventory etc.).
- There are many methods of intercommunicating information between processes. We would also like to have processes on separate machines intercommunicate, too.

1.2 Files

- Files are the most obvious way of passing information around. One process writes a file, and another reads it later. This works, but isn't very interactive – you can't use a file to query a database. It can also be very slow, due to disk speeds. However, it is often used for IPC, especially for time-separated files.
- Question: what problems do files have as an IPC mechanism for concurrent processes?

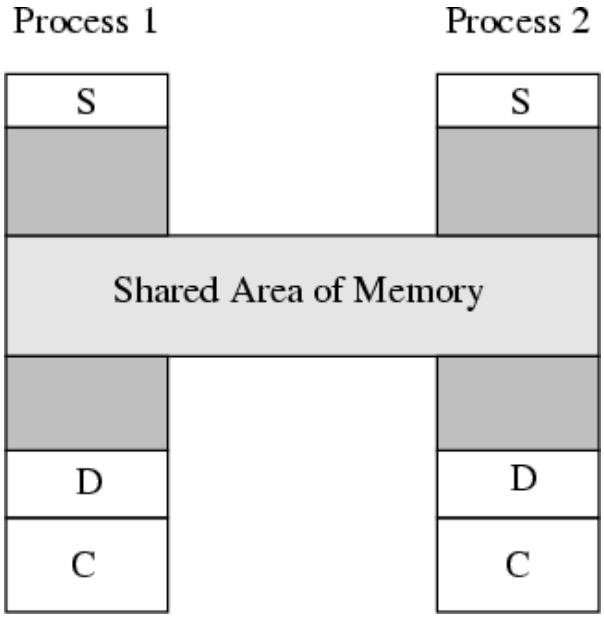
1.3 Pipes – Unidirectional Streams

- We often need to pass output from one process directly to another process. We could write the output to a file and invoke the next process, but this wastes disk space, especially if the file is discarded after the second process exits.
- A classic example is program compilation. In C, the source code is passed through a *preprocessor* before being compiled down to intermediate code by the *compiler front-end*. This is then optimised by the *optimizer*, producing position-independent code. This, finally, has to be linked with any required libraries by the *linker/loader*.
- In this situation, there are several intermediate files which will be deleted once the final binary is produced. Many operating systems provide a method of connecting the output **stream** of data from one process to the input of another; this is known as a **pipe** under Unix. A pipe avoids the use of a temporary file. It is unidirectional, so the second process can't talk back to the first.
- One pipe problem is the size of the pipe, which is usually limited, e.g. 64K under Unix. If the second process isn't reading as fast as the first is writing, the pipe fills. What should be done?
- Unix **blocks** the first process until the second has read some data from the pipe. Similarly, if there is nothing in the pipe, the second process will block reading from the pipe.

- Under some systems, the two ends of a pipe are shared between processes that are related (e.g child processes in Unix). In other systems, the pipe ends can be owned by non-related processes. In this latter case, each pipe end needs a `name' so that processes can find the pipe and connect to it asynchronously.
- The naming scheme is usually dependent on the operating system. Pipe ends may be given dummy file names, and can be opened and used just like files. Alternatively, pipe ends may just be given unique numbers.
- In any case, how does a process know which pipe to attach itself to? Solutions here are to use a well-known name for certain services, or to have a name lookup service to find the service you want. Obviously, the name service must have a well-known name!
- By including the computer's networking address in the pipe's name, IPC via pipes/streams can be performed over networks. This is essentially how most of the Internet's communications works, i.e. TCP, ports and sockets.

1.4 Shared Memory

- This is the *fastest* IPC of the lot. Let the operating system map some pages into the same logical locations in two or more processes.



- This is easy to do with a paged architecture. The operating system already keeps a list of pages that a process owns; just add a few more to the list. However, the operating system needs to keep a **link count** (like for shared files), so that the page can be freed when the link count becomes zero.

2 Synchronisation

2.1 Race Conditions and Critical Sections

- We have seen that processes can use shared memory or files for IPC. This can lead to strange effects when there are concurrent processes using the same resource. For

example, two processes writing to the same file may cause the file to have unknown contents, as one of the writes will itself be overwritten.

- Shared memory is even weirder. Imagine two (or more) accounting processes sharing memory. Each wants to withdraw an amount of money from an account:

```
if (balance - withdrawl >= 0)
   balance = balance - withdrawl;
else
   error("Can't withdraw that much!");
```

- Imagine balance is 800, process A wants to withdraw 500 and process B wants to withdraw 400. If, just after A checks B–W ^{3} 0 and gets 300, it is **pre–empted**, and B is scheduled, B will do the check and withdraw 400, leaving 400. When A is rescheduled, it will already have done the check, and perform the withdrawl, leaving a balance of –100, which is wrong.
- This is a **race condition**, where two or more processes are reading or writing some shared data, and the final result depends on who runs precisely when.
- The section of code where the race condition occurs is a **critical section**; here it is

```
if (balance - withdrawl >= 0)
   balance = balance - withdrawl;
```

- A critical section happens because it is composed of several steps, and a process can be scheduled out after any of the steps. If there was only one **atomic** step, there would be no critical section, as a process could only be scheduled out after the entire operation.
- An operation is atomic if it is guaranteed to be performed all at once, with no interruptions.

2.2 Avoiding a Critical Section

- To avoid a critical section, we need the following conditions:
  1. No two processes can be inside the critical section simultaneously. As we will see, this isn't enough, so we add three more conditions.
  2. No assumptions are made about the relative speeds of each process or the number of CPUs. This disallows solutions based on exact timing.
  3. No process stopped outside its critical section should block another process.
  4. No process should wait arbitrarily long to enter the critical section. In other words, starvation is not allowed.
- Consider a general situation where there is a shared object, e.g the number of stock left for a particular item in a wharehouse, with multiple requests for that stock coming in from various parts of the country. Timing of the requests are unpredictable, and so is the duration of the operations to ship items and update the wharehouse records.
- Let us look at some possible solutions for avoiding critical sections.

2.3 Infinite Timeslices

- Remember, a process is scheduled out when running if its timeslice expires. If the process could obtain an infinite timeslice, there would be no pre-emption. Therefore, it would complete the critical section, and satisfy condition 1.
- However, this is not a good solution, as there is no guarantee that the process will either die, or relinquish the CPU. Thus, this solution violates condition 4.

2.4 Strict Alternation/Rotation

- One solution is, for each process:

```
for (ever) {
  loop while (turn != 0);    /* Wait */    /* != 1, != 2, != 3 etc */
  criticalsection();
  turn= 1;                                 /* =2, =3, =0 etc */
  other stuff;
}
```

- Thus, the access to the critical section rotates through each process.
- This is a **bad** solution, as each process loops continuously waiting for the value of `turn` to change and thus wasting the CPU. This is known as **busy-waiting**.
- Also, the solution relies on rotation through the set of processes. Even if a process doesn't want the critical section, it is given it, and others must wait for it to go through the section, thus violating condition 3. Finally, a slow process slows down the rotation, thus violating condition 2.

2.5 Test and Set Lock Instruction

- If the CPU provides an atomic way of reading and overwriting an address with a `1' (a **test and set lock** instruction), we can create the following:

```
enterregion: tsl register, flag /* Copy the flag to the register, and set flag to 1 */
             cmp register, 0    /* Was flag zero? */
             bnz enterregion    /* No, loop until it is */
leaveregion: mov flag, 0;       /* Set the flag to zero */
             ret                /* and return */
```

- Now we can protect the critical section with:

```
entersection();
/* Critical section stuff */
leaveregion();
```

- A process can only enter the critical section if the `flag` is zero, and in checking the flag, it is set to one, thus preventing anybody else from entering the critical section. This satisfies condition 1.
- This also avoids infinite timeslices and the rotation problem. However, we still have busy-waiting, as we loop until we have the `flag`, which wastes the CPU. At the same time, all four conditions are satisfied.
- To prevent busywaiting, we need a way of **blocking/unblocking** a process if it wants to obtain access to a critical section, but can't get it yet. **Hint:** What normally performs process blocking?

2.6 Semaphores

- Semaphores were invented by Edgar Dijkstra in 1965. The operating system provides new objects called **semaphores**, each with integer values. The operating system also provides two *system call* operations on semaphores:

```
acquire(sem) {
  if (sem.count == 0)
    block process;
  sem.count=0;        /* When reawakened, lower count back to 0 */
}


release(sem) {
  sem.count=1;        /* Raise semaphore's value to 1 */
  if (any process blocked waiting for sem)
    unblock one;
}
```

- The operating system performs both operations in kernel mode, and guarantees that both operations are *atomic*. Now, with the semaphore count initialised by the operating system to 1, we can do:

```
acquire(sem);
/* Critical section stuff */
release(sem);
```

- The first process to acquire the semaphore sets the count to zero. All other processes that try to access the semaphore are blocked. When the process with the semaphore releases it, it sets the count to one,and another process is unblocked. As soon as it is unblocked, it lowers the count to zero, and thus acquires the semaphore.
- This solution satisfies all four conditions, and also avoid busy waiting. Usually, the operating system keeps a queue for the set of processes blocked on a semaphore, but this is not strictly necessary.

2.7 Monitors

- Of course, if a process goes ahead and enters a critical section without using any synchronisation method, problems will occur. We have to trust that all programmers will do the right thing, and also that the code they produce is correct.
- Hoare and Hansen in 1975 suggested building synchronisation into the language so that it is invoked without conscious work by the programmers. The **monitor** is a collection of **code** and condition variables which describe the critical section. The compiler wraps the synchronisation code around the critical section transparently. The synchronisation code can be semaphores, or whatever.
- Not many languages provide monitors. The only one that I can think of off-hand is Java.

2.8 Synchronisation Within the Operating System

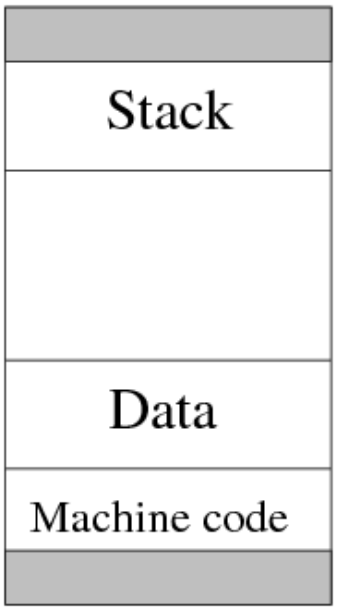
- So far we have looked at synchronisation between processes. However, in many operating systems, different sections of the system may share variables (e.g the process which is running, the free block list, the free memory list).

- Each of these operating system sections may be invoked due to system calls and interrupts (timers, I/O completion, timeslice expiry), and thus may "pre-empt" another section's execution.
- Without protection, the shared variables may be corrupted due to pre-emption in a non-atomic operation (i.e a critical section). However, we can't use semaphores, because the operating system can't block/unblock itself. Fortunately, because the operating system manages the hardware, it can use this to protect the critical sections.
- To protect a critical section, the operating system can *disable* all interrupts. Any interrupts that arrive will be ignored until the operating system re-enables them. This ensures that the execution of the critical section won't be interrupted, and makes the critical section atomic. This might violate condition 4, but we know that the operating system must perform its tasks quickly, so a critical section will not take arbitrarily long to finish.
- It is also a good idea to disable only those interrupts which might cause a critical section; for example, leaving disk interrupts enabled probably won't affect your handling of data delivery from the network.
- We noted the use of interrupt disabling, in the example disk driver from FreeBSD.

3 Threads

3.1 Introduction

- A process is a sequence of instructions executing in an *address space*, with access to the operating system's services. The process consists of: machine instructions, a data area, a stack, and the machine's registers it is using.



- Switching between processes (a **context switch**) is expensive, as the operating system must save/reload the processes' registers and change memory protections.
- In many instances, a process would like to be able to perform several independent tasks that can be performed concurrently. Examples of this are database and other servers, and network protocol implementations.

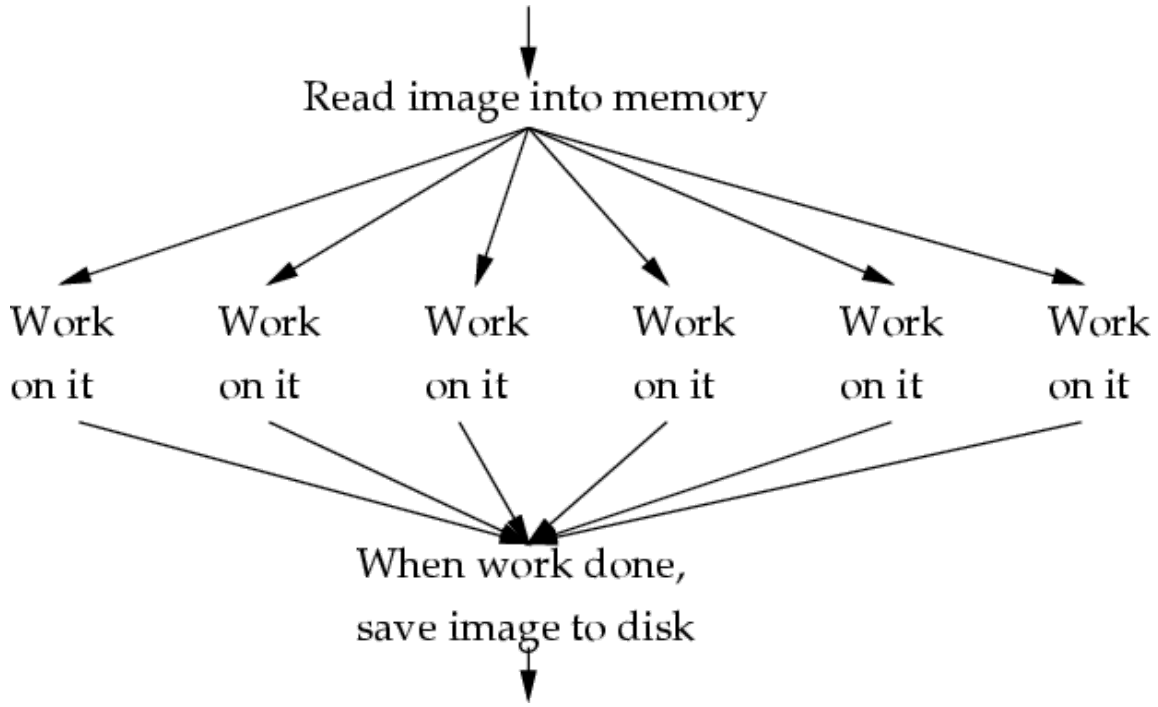
- A specific example, a Web server:

```
loop forever {
  get web page request from client
  check page exists and client has permissions
  transmit web page back to the client
}
```

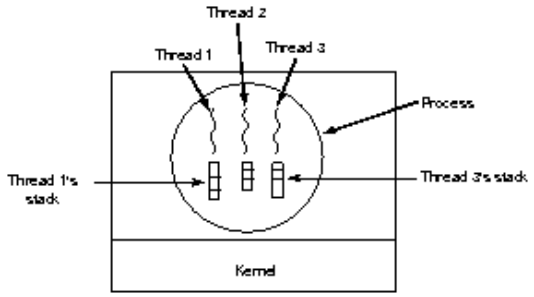
- If the transmission takes a very long time, the server is not able to answer other clients' requests. The server could create a *clone* server to handle the transmission:

```
loop forever {
  get web page request from client
  check page exists and client has permissions
  make a clone of the server
  original server goes back to `get web page'
  clone server transmits web page back to the client
}
```

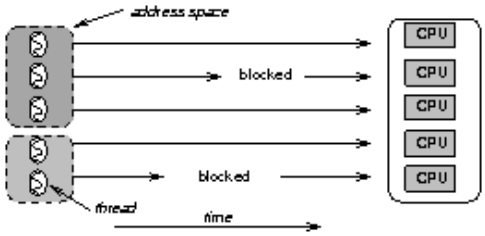
- However, if the new server is another process, there is extra context switching overhead. Also, the original server might cache pages in memory, so it would be useful for the two processes to share memory.
- In other situations, some of the tasks to be performed can be done concurrently, but for one stage there is only one task to do, e.g an image manipulation program:



- A **thread** is a computational unit within a process. Each thread is relatively independent, but may sometimes need to synchronise with other threads. All threads share a common logical address space. A process, therefore, consists of one or more threads.



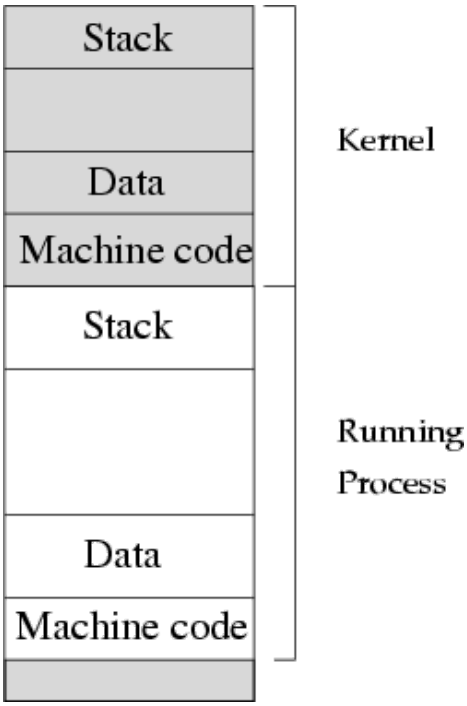
- Advantages of threads: context switching between threads has less overhead because memory maps do not need to be changed. Switching between threads in two *different* processes has normal overhead. Threads share memory, and so can share information to perform their tasks. When running on a multiprocessor machine, each thread can be scheduled to run on a separate CPU, thus increasing performance.



- Disadvantages of threads: threads need to have synchronisation primitives available to prevent deadlocks and other problems. Threads have read/write access to other threads' memory, so a badly-behaved thread can damage other threads.
- There are several ways of implementing threads. Here are some that are currently in use.

3.2 Kernel Threads

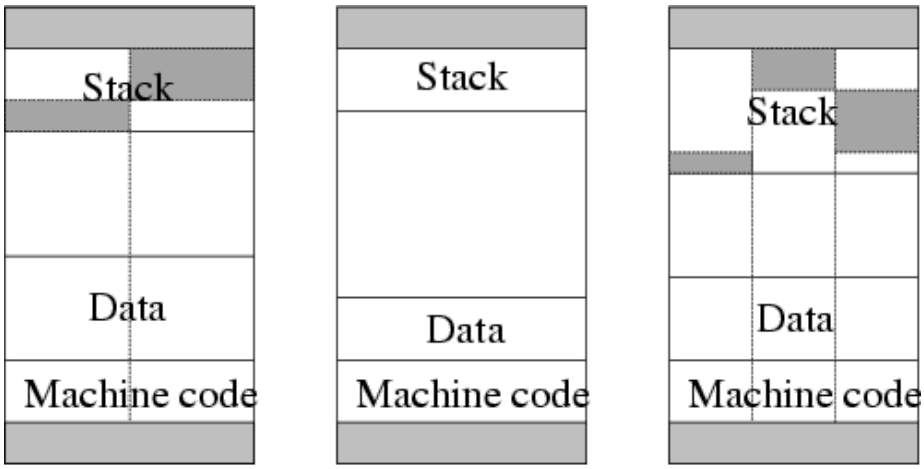
- The operating system can make use of threads to perform tasks such as asynchronous I/O. Instead of providing special mechanisms to do the I/O, the kernel can start a **kernel thread** to perform the work. A good example task that can be handled by kernel threads is to perform network transmissions.
- There is a kernel-mode process (the **kernel process**) which has several kernel threads. Each thread has as it address space all of kernel memory, and all of the running process' memory.



- A kernel thread, therefore, doesn't really have its own memory address space; it borrows the running process' and the kernel's. It does have registers and a stack as its context. Context switching of kernel threads is fast.
- Disadvantages: only the operating system can use kernel threads, as each runs in kernel-mode and the address space is not tied to a particular process.

3.3 Lightweight Processes

- A **lightweight process** is a kernel-supported user-mode thread. Each LWP has its own context; it does not borrow address spaces. Threads within a process have the same address space, and so context switching between them is fast.
- Each thread needs its own stack for local variables, but the entire address space is shared between the threads, giving fast context switches but no inter-stack protection.

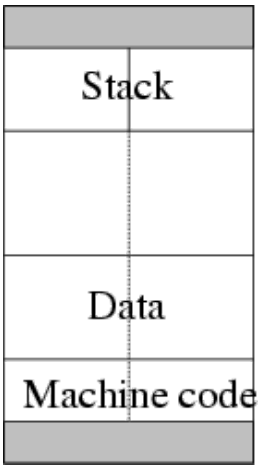


2 Threads in a Process | Single-threaded Process | 3 Threads in a Process

- Each thread can perform normal operations such as read/write memory, make systems calls. A thread is blocked just like a normal process. Other threads within the process can continue execution.
- LWPs need synchronisation operations to protect the shared memory. The operating system provides these operations via system calls. The disadvantage here is that a system call adds a lot of overhead to any operation, and so LWPs that must do frequent synchronisations suffer poor performance. If the critical section is only a few instructions, *busy-waiting* may be preferable than blocking. Similarly, LWP creation/destruction operations are system calls and are expensive.
- As each LWP has a context, the operating system must save this state. The state information is the same size as for a normal process. LWPs are what most people mean when they say `threads'.

3.4 Mediumweight Processes

- A **mediumweight process** is a lightweight process but with a separate stack from the other MWPs. Machine code and global data is still shared.



- The advantage is inter-stack protection, but a MWP context switch must perform more memory re-adressing, so it is slower. Traditional processes are often known as **heavyweight processes**.

3.5 User Threads

- It is possible to provide threads on a system where the kernel knows only about traditional processes. This is accomplished through library packages such as Mach's *C-threads* and POSIX *pthreads*. The libraries provide the functions for thread creation/destruction and synchronisation, with no help required from the kernel.
- The library takes responsibility for saving the registers when performing `internal' context switches between threads in a process. The kernel only performs context switches between normal processes.
- Because the kernel is not involved, thread context switching and synchronisation is very fast. Also, there are no extra resources required from the operating system to support user threads. However, if one thread in a process is blocked by the operating system, all

the threads in the process are blocked. If the system has multiple processors, threads cannot be executing on more than one CPU.

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 14:29.