Scheduling with several CPUs

The term **multiprocessing** refers to **multiple processors** not multiple processes. Likewise, **multiprocessing** means using multiple CPUs.

Multiprocessor systems are divided into tightly–coupled and loosely–coupled systems depending on whether they share memory or not.

Multiprocessing is called **asymmetric** when one processor had a different function than the others; when all the processors are equal, multiprocessing is called **symmetric**.

Early history

In 1961, Burroughs Corporation introduced the first symmetric multiprocessor (Burroughs 5500) with four CPUs and up to sixteen memory modules connected via a crossbar switch (the first SMP architecture).

The popular and successful CDC 6600 was introduced in 1964 and provided a CPU with ten subprocessors (peripheral processing units) one of which a dedicated processor running part of the OS code (so it was asymmetric).

In the late 1960s, Honeywell delivered the first Multics system, another symmetric multiprocessing system of eight CPUs.

All these systems were tightly coupled.

Servers

Multiprocessing systems became less popular for a while. They reemerged as network–based servers in the form of loosely–coupled clusters, i.e. semi–independent computers connected by a high–speed interconnect (network). Clusters became fairly popular in the 1990s.

The main limitation of clusters was the communication part, a single high–speed channel. It cannot deliver a bandwidth similar to that of a memory bus as used in a tightly–coupled system. Nevertheless, loosely–coupled systems persist (e.g. Linux Beowulf clusters).

The introduction of multi–core processors gave a boost to tightly–coupled multiprocessing and required operating systems that could manage several processors.

SMP Scheduling

Nowadays, all tightly–coupled multiprocessor systems use **SMP** or **Symmetric MultiProcessing** an approach which treats all the **CPU**s equally.

Scheduling processes in a tightly–coupled system is not complex: the memory being shared, that a process can use any CPU without noticing that it is not the same CPU as before. This makes **process migration** (from CPU to CPU) easy.

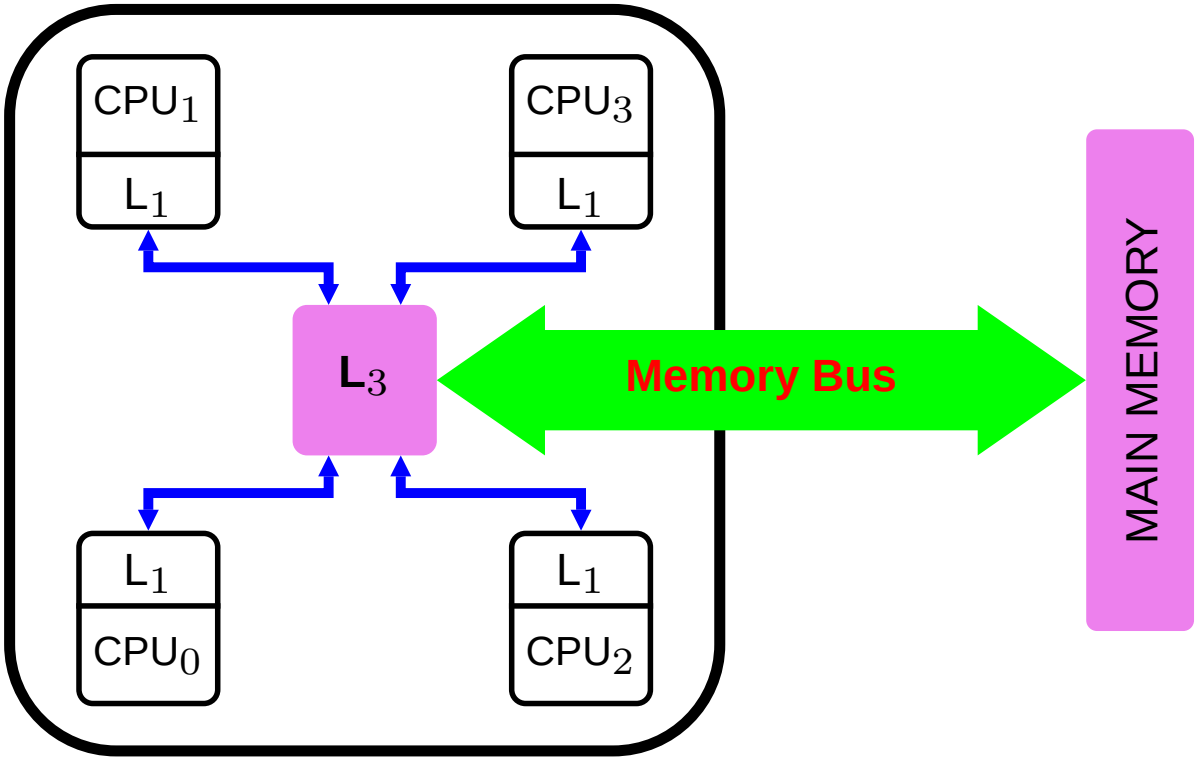
Alas, hardware designers are under pressure to make the hardware appear faster and faster. To achieve that, they keep adding features that reduce the time to execute a sequence of machine instructions.

These additions have an adverse effect on OS scheduling because the essence of all the additions is to bring all the needed memory as close to the CPU as possible. That implies that that memory is taken farther away from the other CPUs.

Scheduling on an SMP system introduces a new consideration: the cost of dispatching a process in a different CPU. This cost is not quite clear and is architecture dependent. With architectural changes made yearly it is hard to assess the usefulness of SMP for process scheduling.

Multicore architectures

A typical multi–core system is made of a number of **CPU**s, each with its own cache (L1, also L2 in i7), interconnected via a shared bus to a common high–speed cache memory (L3 cache, earlier L2) and, through it, to a common main memory.



The existence of private cache memories makes process migration (from CPU to CPU) very costly. To reduce the overhead of flushing a cache, the concept of processor affinity was introduced. Although affinity is supposedly expressed as a form of virtual gravity that can be overcome by a superior force (pull or push), it is implemented (in Linux) as a unconditional attachment to a particular CPU.

Affinity can be enforced at the process level (e.g. a process wants to run only on CPU3) or at the interrupt level (e.g. IRQ 8 can only be executed on CPU1 or CPU2).

IRQ affinity

Linux systems allow the user to limit processing of specific interrupt types (**IRQ**s) to a subset of all the available CPUs. This has the effect of forcing parts of the OS to execute on a subset of all the CPUs.

This is done by putting in file `/proc/irq/24/smp_affinity` a mask naming the CPUs allowed to process IRQ 24 (replace 24 with your favourite number if desired).

The mask is:

CPU 0	0001	1
CPU 1	0010	2
CPU 2	0100	4
CPU 3	1000	8

So, a mask of 6 names CPUs 1 and 2.

Threads and parallelism

In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers.

Recently, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

Pthreads come in a package called **pthreads** for UNIX–based systems. Win32 versions exist but seem incomplete ((click here for details).

When you use pthreads you will need to specify that threads are used by giving a compiler flage, such as:

- gcc -pthread code.c
- gcc code.c -lthreads
- or similar.

Some references:

- Tutorial from LLNatLab: all the details, lots of colours and code.
- YoLinux tutorial: lots of informative code.
- Another tutorial
- LUPG tutorial, long

These references contain working code written in **C**.

What is a thread

A **thread** is a portion of a process, a semi–process (another term is **lightweight process**) that has its own stack, and executes a given piece of code. Unlike a real process, the thread shares its global variables with other threads (where as for processes we usually have a different memory area for each one of them).

A **Thread Group is a set of threads** all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory (malloc()), same set of file descriptors, etc.

All these threads execute concurrently (using time slices) or in parallel, if the system has several **CPU**s.

The pthreads **API** combines shared memory and semaphores into one set of functions. Sadly, the standard uses different names for these functions than the ones used in **POSIX XSI** shared memory and semaphore standard.

Example

Suppose we have the urge to execute the function do_loop twice.

```
void do_loop(int me)
{
int i;        /* counter, to print numbers */
int j;        /* counter, for delay        */

   for (i=0; i<10; i++) {
      for (j=0; j<50000 ; j++) /* delay loop */
         ;
      printf("'%d' - Got '%d'\n", me, i);
   }
   exit(0);
}
```

I can fork a process

```
// parent process starts execution in main


int main(int argc, char* argv[])
{
pid_t child;        // pid of the newly created child


   if( (pid = fork()) == 0 ) {
      do_loop( 1 ) ;
   else
      do_loop( 2 );


   printf( "If you see this message, say
huh?\n" ) ;
}
```

Same using threads

The code changes because the pthread_create system call requires specific argument types.

```
// execution begins in main (single thread starts)


int main(int argc, char* argv[])
{
int thr_id;        // thread ID for the newly created thread
pthread_t  p_thread;    // thread's structure
int  a = 1;  // thread 1 identifying number
int  b = 2;  // thread 2 identifying number

   thr_id = pthread_create(&p_thread, NULL, do_loop,
(void*)&a);
   do_loop((void*)&b);

   printf( "If you see this message, say
huh?\n" ) ;
}
```

Must be compiled using **cc -pthread** or **gcc -pthread**.

pthread_create() has 4 arguments:

- The first is used by pthread_create() to return to the program information about the thread.
- The second is used to set some attributes for the new thread. In our case we supplied a NULL pointer to tell pthread_create() to use the default values.
- The third is the name of the function that the thread will start executing. It must return a void *.
- The fourth is an argument (or argument list) to pass to the function. It must be of type void *.

The function must be rewritten to match pthread_create:

```
void* do_loop(void* data)
{
int i;        /* counter, to print numbers */
int j;        /* counter, for delay        */
int me = *((int*)data);     /* thread identifying number */

   for (i=0; i<10; i++) {
      for (j=0; j<50000 ; j++) /* delay loop */
         ;
      printf("'%d' - Got '%d'\n", me, i);
   }
   pthread_exit( NULL ) ;
}
```

pthread_exit() terminates the thread (note that the main process is a thread, too, so it also terminates with a pthread_exit).

Memory sharing

All the **pthreads** forming one group share all their global memory. That includes the memory placed on the **heap** (i.e. acquired using malloc).

Whenever a thread calls a function, the local variables of that function land on the private stack of the calling thread. They are not accessible by other threads. In the do_loop function there will be two sets of private variables i, j, me; one set for each thread. These variables will be different and hence will have different values in each thread.

Semaphores in threads

pthreads have semaphores which are called **mutex**es.

A mutex is declared like any other **global** variable:

```
pthread_mutex mutex = PTHREAD_MUTEX_INITIALIZER ;
```

This gives a properly initialised semaphore.

The operations on these semaphores are very simple:

P(mutex): `pthread_mutex_lock( &mutex ) ;`

V(mutex): `pthread_mutex_unlock( &mutex ) ;`

There is much more

pthreads support many other features:

- Waiting for events to happen or conditions to become true.
- Joining threads (like a return).
- Thread cancellation.