Chapter 3

Multicore and Multiprocessor Systems: Part I

Symmetric Multiprocessing

Definition (Symmetric Multiprocessing (SMP))

The situation where two or more identical processing elements access a shared periphery (i.e., memory, I/O,...) is called *symmetric multiprocessing* or simply (SMP).

The most common examples are

- Multiprocessor systems,
- Multicore CPUs.

Memory Hierarchy

Basic Memory Layout

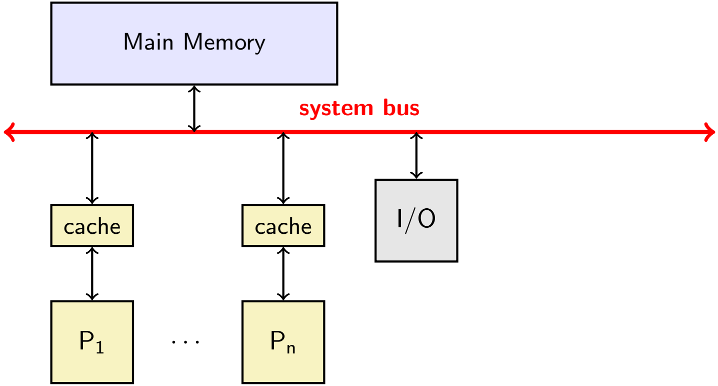

Figure: Schematic of a general parallel system

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,
- is shared among all processing units,

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Especially that means that all memory locations can be requested by all processors at the same time scale, independent of which processor preforms the request and which chip in the memory holds the location.

Memory Hierarchy

Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Especially that means that all memory locations can be requested by all processors at the same time scale, independent of which processor preforms the request and which chip in the memory holds the location.

Local caches one the single processing units are allowed. That means classical multicore chips are an example of a UMA system.

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one logical shared memory unit,

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one logical shared memory unit,
- gathered from two or more physical resources,

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Due to the distributed nature of the memory, access times vary depending on whether the request goes to local or foreign memory.

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Due to the distributed nature of the memory, access times vary depending on whether the request goes to local or foreign memory.

Examples are current multiprocessor systems with multicore processors per socket and a separate portion of the memory controlled by each socket. Also recent "cluster on a chip" design processors like AMDs bulldozer

Memory Hierarchy

Non-Uniform Memory Access (NUMA)

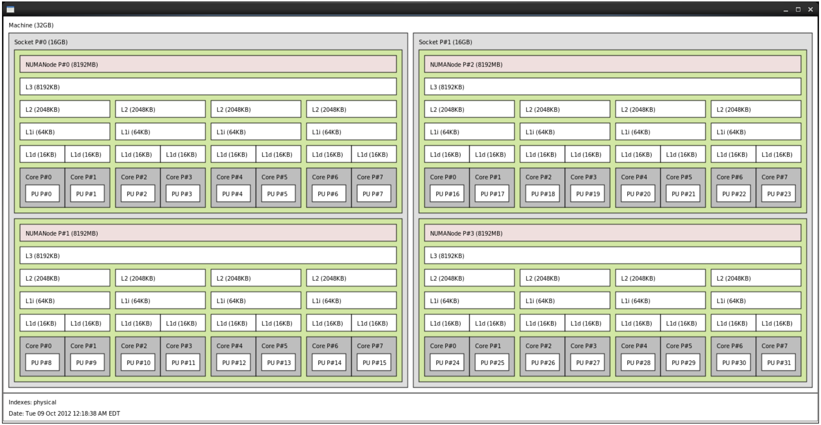

Figure: AMDs Bulldozer layout is a NUMA example.[1]

[1] By The Portable Hardware Locality (hwloc) Project (Raysonho@Open Grid Scheduler / Grid Engine) [see page for license], via Wikimedia Commons

Memory Hierarchy

Cache Coherence

Definition (cache coherence)

The problem of keeping multiple copies of a single piece of data in the local caches of the different processors that hold it consistent is called cache coherence problem.

Cache coherence protocols:

- guarantee a consistent view of the main memory at any time.
- Several protocols exist.
- Basic idea is to invalidate all other copies whenever one of them is updated.
- Compare also seminar talk by Carolin Penke entitled “Cache coherent Non-Uniform Memory Access (architecture/difficulties/tools)”

Processes and Threads

Multiprocessing

Definition (Process)

A computer program in execution is called a process.

A process consists of:

- the programs machine code,
- the program data worked on,
- the current execution state, i.e., the context of the process, register and cache contents, ...

Each process has a separate address space in the main memory.

Execution time slices are assigned to the active processes by the operating systems (OSs) scheduler. A switch of processes requires exchanging the process context, i.e., a short execution delay.

Processes and Threads

Multiprocessing

Multiple processes may be used for the parallel execution of compute tasks.

On Unix/Linux systems the `fork()` system call can be used to generate child processes. Each child process is generated a copy of the calling parent process. It receives an exact copy of the address space of the parent and a new unique process ID (PID).

Communication between parent and child processes can be implemented via sockets or files, which usually leads to large overhead for data exchange.

Processes and Threads

Threading

Definition (Thread)

In the thread model a process may consist of several execution sub-entities, i.e, control flows, progressing at the same time. These are usually called threads, or lightweight processes.

All threads of a process share the same address space.

Processes and Threads

Threading

Two types of implementations exist:

- **user level threads:**
  - administration and scheduling in user space,
  - threading library maps the threads into the parent process,
  - quick task switches avoiding the OS.
- **kernel threads:**
  - administration and scheduling by OS kernel and scheduler,
  - different threads of the same process may run on different processors,
  - blocking of single threads does not block the entire process,
  - thread switches require OS context switches.

Processes and Threads

Threading

Two types of implementations exist:

- **user level threads:**
  - administration and scheduling in user space,
  - threading library maps the threads into the parent process,
  - quick task switches avoiding the OS.
- **kernel threads:**
  - administration and scheduling by OS kernel and scheduler,
  - different threads of the same process may run on different processors,
  - blocking of single threads does not block the entire process,
  - thread switches require OS context switches.

Here we concentrate on POSIX threads, or Pthreads. These are available on all major OSes. The actual implementations range from from user space wrappers (`pthreads-w32` mapping pthreads to windows threads) to lightweight process type implementations (e.g. Solaris 2).

Processes and Threads

Mapping of user level threads to kernel threads or processes

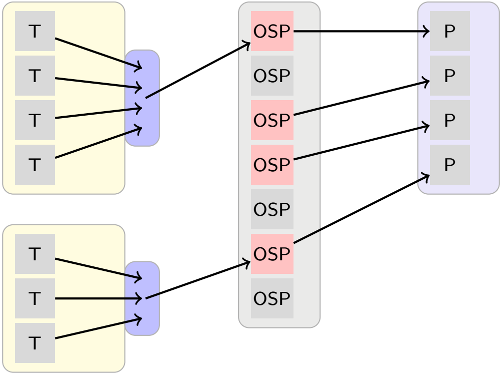

Figure: N:1 mapping for OS incapable of kernel threads

Processes and Threads

Mapping of user level threads to kernel threads or processes

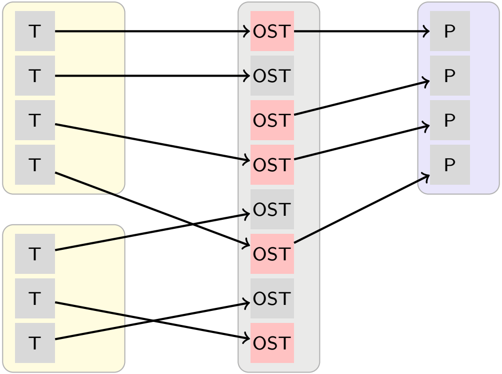

Figure: 1:1 mapping of user threads to kernel threads

Processes and Threads

Mapping of user level threads to kernel threads or processes

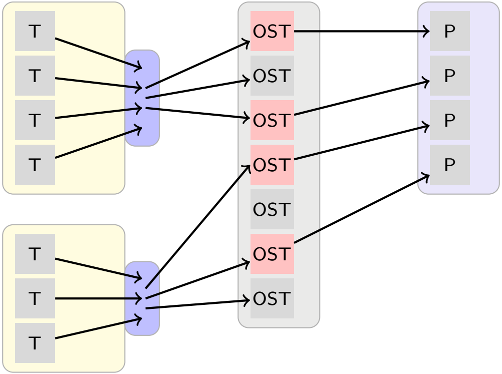

Figure: N:M mapping of user threads to kernel threads with library thread scheduler

Processes and Threads

Properties and Problems

Parallel versus concurrent execution

1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.

Processes and Threads

Properties and Problems

Parallel versus concurrent execution

1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.

Processes and Threads

Properties and Problems

Parallel versus concurrent execution

1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.
3. The concurrent execution only requires a quasi parallel environment that allows all tasks to be in progress at the same time.

Processes and Threads

Properties and Problems

Parallel versus concurrent execution

1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.
3. The concurrent execution only requires a quasi parallel environment that allows all tasks to be in progress at the same time.
4. That means "parallel" execution defines a subset of "concurrent" execution.

Processes and Threads

Properties and Problems

Definition (race condition)

When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this leads to a race condition, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.

Processes and Threads

Properties and Problems

Definition (race condition)

When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this leads to a race condition, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.

Example

Thread 1	Thread 2	value
		0
`read`		0
`increment`		0
`write`		1
	`read`	1
	`increment`	1
	`write`	2

Processes and Threads

Properties and Problems

Definition (race condition)

When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this leads to a race condition, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.

Example

Thread 1	Thread 2	value
		0
`read`		0
`increment`		0
`write`		1
	`read`	1
	`increment`	1
	`write`	2

Thread 1	Thread 2	value
		0
`read`		0
	`read`	0
`increment`		0
`write`		1
	`increment`	1
	`write`	1

Processes and Threads

Protection of critical regions

Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Processes and Threads

Protection of critical regions

Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Definition (mutual exclusion variable (mutex))

The mutual exclusion variable or shortly mutex variable implements a simple locking mechanism regarding the critical region. Each process upon entry to the region checks the lock. If it is open the process/thread enters and locks it behind. Thus all other processes/threads are prevented from entering and the process in the critical region has exclusive access to the shared data. When exiting the region the lock is opened.

Processes and Threads

Protection of critical regions

Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Definition (mutual exclusion variable (mutex))

The mutual exclusion variable or shortly mutex variable implements a simple locking mechanism regarding the critical region. Each process upon entry to the region checks the lock. If it is open the process/thread enters and locks it behind. Thus all other processes/threads are prevented from entering and the process in the critical region has exclusive access to the shared data. When exiting the region the lock is opened.

Both the above definitions introduce the programming models. Actual implementations may be more or less complete. For example the `pthreads`-implementation lacks counting semaphores.

Processes and Threads

Protection of critical regions

deadlock

A deadlock describes the unfortunate situation, when semaphores, or mutexes have not or inappropriately been applied such that no process/thread is able to enter the critical region anymore and the parallel program is unable to proceed.

Processes and Threads

Dining Philosophers

Example (dining philosophers)

Figure: The dining philosophers problem

- Each philosopher alternatingly eats or thinks,
- to eat the left and right forks are both required,
- every fork can only be used by one philosopher at a time,
- forks must be put back after eating.

Processes and Threads

Dining Philosophers

simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

Processes and Threads

Dining Philosophers

simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

All philosophers decide to eat at the same time $\Rightarrow$ deadlock.

Processes and Threads

Dining Philosophers

simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

All philosophers decide to eat at the same time $\Rightarrow$ deadlock.

More sophisticated solutions avoiding the deadlocks have been found since [DIJSTRA '65]. Three of them are also available on Wikipedia[a].

[a] http://en.wikipedia.org/wiki/Dining_philosophers_problem

POSIX Threads

Basics

Common to all the following commands:

Compiling and linking needs to be performed with `-pthread`.

The pthread functions and related data types are made available via:

```
#include <pthread.h>
```

POSIX Threads

Creation of threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- `thread` unique identifier to distinguish from other threads,
- `attr` attributes for determining thread properties. `NULL` means default properties,
- `start_routine` pointer to the function to be started in the newly created thread,
- `arg` the argument of the above function.

POSIX Threads

Creation of threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- `thread` unique identifier to distinguish from other threads,
- `attr` attributes for determining thread properties. `NULL` means default properties,
- `start_routine` pointer to the function to be started in the newly created thread,
- `arg` the argument of the above function.

Note that only a single argument can be passed to the threads start function.

POSIX Threads

Creation of threads: multiple arguments of the start function

The argument of the start function is a `void` pointer. We can thus define:

```
struct point3d{
  double x,y,z;
};
struct point3d P;
```

and upon thread creation pass

```
err=pthread_create(tid, NULL, norm, (void *) P);
```

to a start function

```
void *norm(void *arg) {
  return arg.x*arg.x + arg.y*arg.y + arg.z*arg.z;
};
```

POSIX Threads

Creation of threads: Possible race conditions

```
int main(int argc, char* argv[]){
  pthread_t tid1,tid2;

  point3d point;

  point.x=10; point.y=10; point.z=0;
  pthread_create(&tid1, NULL, norm, &point);
  point.x=20; point.y=20; point.z=-50;
  pthread_create(&tid1, NULL, norm, &point);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

Depending on the execution of thread `tid1` the argument `point` may get overwritten before it has been fetched.

POSIX Threads

Exiting threads and waiting for their termination

Pthreads can exit in different forms:

POSIX Threads

Exiting threads and waiting for their termination

Pthreads can exit in different forms:

- they return from their start function,

POSIX Threads

Exiting threads and waiting for their termination

Pthreads can exit in different forms:

- they return from their start function,
- they call `pthread_exit()` to cleanly exit,

POSIX Threads

Exiting threads and waiting for their termination

Pthreads can exit in different forms:

- they return from their start function,
- they call `pthread_exit()` to cleanly exit,
- they are aborted by a call to `pthread_cancel()`,

POSIX Threads

Exiting threads and waiting for their termination

Pthreads can exit in different forms:

- they return from their start function,
- they call `pthread_exit()` to cleanly exit,
- they are aborted by a call to `pthread_cancel()`,
- the process they are associated to is terminated by an `exit()` call.

POSIX Threads

Exiting threads and waiting for their termination

```
int pthread_exit(void *retval);
```

- `retval` return value of the exiting thread to the calling thread,
- threads exit implicitly when their start function is exited,
- the return value may be evaluated from another thread of the same process via the `pthread_join()` function,
- after the last thread in a process exits the process terminates calling `exit()` with a zero return value. Only then shared resources are released automatically.

POSIX Threads

Exiting threads and waiting for their termination

```
int pthread_join(pthread_t thread, void **retval);
```

- Waits for a thread to terminate and fetches its return value.
- `thread` the identifier of the thread to wait for,
- `retval` destination to copy the return value (if not `NULL`) to.