Introduction to OpenMP

Edgar Gabriel
Fall 2011

Introduction

- Threads vs. processes
- Recap of shared memory systems
  - SMP vs. NUMA
  - Cache Coherence
  - Process and thread affinity
- Operating system scheduling
  - Load balancing across multiple processors
Processes vs. Threads

• Process:
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads
  - a program that is running

• Thread:
  - a sequence of instructions executed within a process
  - shares the resources in that process

Processes vs. Threads (II)

• Advantages of multi-threaded programming:
  • easier programming than multi-process models
  • lower overhead for creating a thread compared to creating a process
  • lower overhead for switching between threads required by the OS compared to switching between processes

• Drawbacks of multi-threaded programming:
  • more difficult to debug than single threaded programs
  • for single processor machines, creating several threads in a program may not necessarily produce an increase in performance
**Processes vs. Threads (III)**

<table>
<thead>
<tr>
<th>Data per process</th>
<th>Data per thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Open files</td>
<td>Processor register</td>
</tr>
<tr>
<td>Child processes</td>
<td>Processor status</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Signal mask</td>
</tr>
<tr>
<td>Timer</td>
<td>Stack</td>
</tr>
<tr>
<td>Accounting</td>
<td></td>
</tr>
</tbody>
</table>

**Execution model**

- **Main thread:**
  - initial thread created when `main()` (in C) invoked by the process loader
  - once in `main()`, the application can create additional threads
  - `exit()` (or `return()` called in the main function) will terminate all threads
    - E.g. the process terminates even if there are still running threads
    - Threads can be terminated separately by using different functions
Threads vs. processes

- In Linux: no difference on the OS level between processes and threads
  - Threads are processes which have access to the same resources, e.g. address space etc.
  - E.g. internally Linux creates a thread by calling
    \[
    \text{clone(CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND,0)};
    \]
- In contrast, a new process is created internally by Linux calling
  \[
  \text{clone(SIGCHLD,0)}
  \]

Recap: shared memory systems

- All processes have access to the same address space
  - PC with more than one processor
  - PC with multi-core processor
- Data exchange between processes by writing/reading shared variables
- Two versions of shared memory systems available
  - Symmetric multiprocessors (SMP)
  - Non-uniform memory access (NUMA) architectures
Symmetric multi-processors (SMPs)

- All processors share the same physical main memory

  ![Diagram of SMP architecture]

- Memory bandwidth per processor is limiting factor for this type of architecture

NUMA architectures (I)

- Some memory is closer to a certain processor than other memory
  - The whole memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly

  ![Diagram of NUMA architecture]
NUMA architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures

Cache Coherence

- Real-world shared memory systems have caches between memory and CPU
- Copies of a single data item can exist in multiple caches
- Modification of a shared data item by one CPU leads to outdated copies in the cache of another CPU
Cache coherence (II)

- Typical solution:
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, ‘notification’ of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- Cache coherence only an issue in case multiple tasks access the same item
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

Thread and Process Affinity

- Each thread/process has an *affinity mask*
  - Specifies what processors a thread is allowed to use
  - Different threads can have different masks
  - Affinities are inherited across process creation
- Example: 4-way multi-core

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>core 3</td>
<td>core 2</td>
<td>core 1</td>
<td>core 0</td>
</tr>
</tbody>
</table>

Process/thread is allowed to run on cores 0,2,3, but not on core 1.

Linux Kernel scheduler API

Retrieve the current affinity mask of a process

```c
#include <sys/types.h>
#include <sched.h>
#include <unistd.h>
#include <errno.h>

unsigned int len = sizeof(cpu_set_t);
cpu_set_t mask;
pid_t pid = getpid();/* get the process id of this app */

ret = sched_getaffinity (pid, len, &mask);
if ( ret != 0 )
    printf("Error in getaffinity %d (%s)\n", errno, strerror(errno);
for (i=0; i<NUMCPUS; i++) {
    if ( CPU_ISSET(i, &mask) )
        printf("Process could run on CPU %d\n", i);
}
```

Linux Kernel scheduler API (II)

Set the affinity mask of a process

```c
unsigned int len = sizeof(cpu_set_t);
cpu_set_t mask;
pid_t pid = getpid();/* get the process id of this app */

/* clear the mask */
CPU_ZERO (&mask);

/* set the mask such that the process is only allowed to
execute on the desired CPU */
CPU_SET ( cpu_id, &mask);

ret = sched_setaffinity (pid, len, &mask);
if ( ret != 0 ) {
    printf("Error in setaffinity %d (%s)\n", errno, strerror(errno);
```
Linux Kernel scheduler API (III)

- Setting affinity mask of a thread

```c
#define __USE_GNU

pthread_setaffinity_np(thread_t t, len, mask);
pthread_attr_setaffinity_np(thread_attr_t a, len, mask);
```

- First function modifies the affinity mask of an existing thread
- Second function sets the affinity mask of a thread before it is created
  - A thread inherits the affinity mask of the main thread and will run on the same core initially as the main thread otherwise

```c
#ifndef _GNU_SOURCE
#define _GNU_SOURCE 1
#endif

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int tid;
thread_attr_t attr;
CPU_ZERO(&cpuset);
CPU_SET(1, &cpuset); // thread will be allowed to run on core 1 only
pthread_attr_init(&attr);
pthread_attr_setaffinity_np(&attr, sizeof(cpuset), &cpuset);
pthread_create(&tid, &attr, worker, (void *)(intptr_t) i);
```
Thread affinity

- E.g. Intel Tigerton Processor

- For threads that have to often access shared data items, costs of update operations are smaller if threads are ‘close to each other’
- Cache coherence effects can be limited if threads are close to each other

Process scheduler

- Component of the kernel that selects which process to run next
- Processes are either waiting to run (blocked) or running
- Multi-tasking: interleave the execution of more than one process
  - Cooperative multitasking: scheduler does not stop a running process until it voluntarily decides to do so (yielding)
  - Preemptive multitasking: scheduler decides when to cease a running process
    - timeslice: amount of time a process is run before being preempted
Process scheduler

- Scheduling policy must satisfy two conflicting goals:
  - Process response time (low latency)
  - Maximum system utilization (high throughput)
- Priority based scheduling in Linux:
  - Rank processes based on their worth for need for processing time
  - Processes with higher priority run before those with lower priority.
  - Processes with the same priority are scheduled round-robin
  - Higher priority processes in Linux receive larger timeslices

Linux dynamic priority based scheduling

- Static priority: also called nice value
  - Nice value ranges from -20 to 19, default is 0,
    - larger nice values correspond to lower priority (you are nice to other processes)

<table>
<thead>
<tr>
<th>Type of Task</th>
<th>Nice value</th>
<th>Timeslice Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially created</td>
<td>Parent’s process</td>
<td>Half of parent’s</td>
</tr>
<tr>
<td>Minimum priority</td>
<td>+19</td>
<td>5ms (MIN_TIMESLICE)</td>
</tr>
<tr>
<td>Default priority</td>
<td>0</td>
<td>100ms (DEF_TIMESLICE)</td>
</tr>
<tr>
<td>Maximum</td>
<td>-20</td>
<td>800ms (MAX_TIMESLICE)</td>
</tr>
</tbody>
</table>

- Dynamic priority value uses the static priority value + ‘interactivity’ of the task
SMP support in Linux

- Linux maintains one *runqueue* per processor
- Each *runqueue* contains two priority queues:
  - *active queues*: list of tasks to be executed
  - *expired queues*: list of tasks that have been executed
  - A task is moved from the *active* to the *expired* queue after execution
  - When a task is moved from the active to the expired queue, dynamic priority and *timeslice* length are recalculated
  - After all tasks in the active queue have executed, both queues are swapped

SMP support in Linux

- Each priority array contains one queue of runnable processes per (dynamic) priority level
  - Interactive tasks can be reinserted into the active queue to run more frequently
  - Interactivity is determined by taking the amount of time that a process sleeps within its *timeslice* into account
    - The more a process sleeps, the more it is driven by I/O operations, hence interactive
Load balancing in Linux kernel

- Load balancer tries to ensure that the number of processes is evenly distributed across the different runqueues (= processors)
- Load balancing function called periodically on all processors
- Load balance algorithm:
  - check all other runqueues for busiest runqueue, i.e. runqueue with the largest number of processes
  - If no runqueue with more than 25% more processes than on the own processors found, exit
  - Else find tasks that can be migrated to own runqueue

Load balancing in Linux

- To migrate a task from one runqueue to another:
  - Search in highest priority queues first on the busiest runqueue
  - Preferably tasks from the expired queue
    - These tasks are not running, and are least likely to have data in the cache
  - Need to take affinity setting of task/process into account
    - Is the process/thread allowed to run on another processor?
- A processor ‘steals’ work from other processors, it never ‘pushes’ work to other runqueues
What does it mean for multi-threaded programs on Linux

- Threads = Processes for Linux
- Whenever a new thread is created:
  - Inherits parent’s process ( = main thread) priority value
  - Inherits half of parent’s process timeslice
  - Has the same affinity mask as the parent process (unless explicitly changed by user)
  - Is most probably in the same runqueue ( = on the same processor) as the parent process
- Once thread has been moved to the expired queue
  - Timeslice for each thread is recalculated ( = increased)
  - Other processors can steal a thread