Thread Scheduling for Multiprogrammed Multiprocessors

Nimar S. Arora Robert D. Blumofe C. Greg Plaxton

Mattias Eriksson — mater@ida.liu.se

16th March 2007

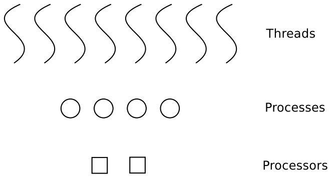
Traditional thread scheduling

- Not multiprogrammed
- Dedicated processors
- Threads dynamically mapped

Multiprogrammed scheduling

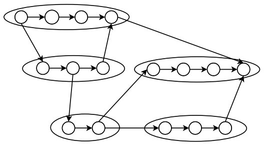
- The processors are not dedicated
- Number of available processors varies over time
- We can not control it

Two levels of scheduling



- *User level:* Threads mapped to processes
- *Kernel level:* Processes mapped to current processor set

The model of the program



- A dag
- T_1
- T_∞
- $\frac{T_1}{T_\infty}$
- $\mathcal{P}$, the set of processes
- $P = |\mathcal{P}|$, number of processes

The model of execution

- Synchronous
- Time steps

A kernel schedule:

$$ks : \mathbb{N} \rightarrow 2^{\mathcal{P}}$$

$$p_i = |ks(i)|$$

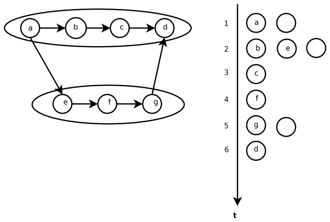
Processor average over T time steps:

$$P_A = \frac{1}{T} \sum_{i=1}^{T} p_i$$

Execution schedule

- which instructions are executed at each time step
- determined by both schedulers!
- the length is defined as T

Example:

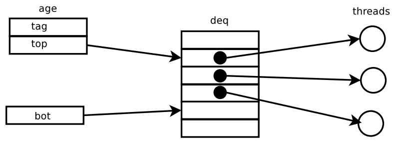


Work stealing user level scheduler

```
/∗ On every process ∗/
Thread ∗thread = NULL;
if (myRank == 0)
  thread = rootThread;

while(!computationDone){
  while(thread != NULL){
    /∗ all spawns are pushed on bottom ∗/
    dispatch(thread);
    thread = self->popBottom();
  }
  /∗ no more work, become THIEF ∗/
  yield(); /∗ but first, give up the cpu ∗/
  Process ∗victim = randomProcess();
  thread = victim->deque.popTop();
}
```

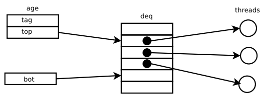
The deque



- One for every process
- Concurrent access → synchronization
- Lock-free implementation with `cas` (atomic)

```
cas(word *addr, word *old, word *_new)
{
  if(*addr == *old)
    SWAP(addr, _new); /* success! (*old == *_new) */
  else
    *_new = *addr; /* failure! */
}
```

Deque operations



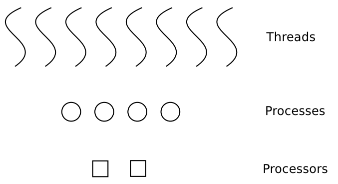
```
Thread *popTop()
{
  oldAge = age;
  localBot = bot;
  if (localBot <= oldAge.top)
    return NULL; /* empty */
  thr = deq[oldAge.top];
  newAge = oldAge;
  newAge.top++;
/*make sure thr is still ok */
  cas(&age,&oldAge,&newAge);
  if(oldAge == newAge)
    return thr;
  /* popTop() can fail */
  return ABORT;
}
```

There is also `popBottom()` and `pushBottom()`

The kernel is an adversary

Three kinds of kernels:

- The benign adversary chooses p_i for each i
- The oblivious adversary chooses both p_i and which processes to execute offline
- The adaptive adversary does the same thing online



We use the yield() system call to influence the kernels' choice of processes

---

Execution time

In the presence of an adversary and yield()

$$E[T] = O(\frac{T_1}{P_A} + \frac{T_\infty P}{P_A})$$

And for $\epsilon > 0$:

$$T = O(\frac{T_1}{P_A} + (T_\infty + \log(\frac{1}{\epsilon}))\frac{P}{P_A})$$

with probability at least $1 - \epsilon$

→ linear speedup when $P << \frac{T_1}{T_\infty}$

Conclusion

> *[...]the non-blocking work stealer executes with guaranteed high performance in [multiprogrammed] environments. [...] the non-blocking work stealer executes any multithreaded computation with work* T_1 *and critical-path length* T_∞*, using any number* P *of processes, in expected time* $O(T_1/P_A + T_\infty P/P_A)$*, where* P_A *is the average number of processors on which the computation executes.*