Thread Scheduling for Multiprogrammed Multiprocessors

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

A summary.[*]

Mattias Eriksson
mater@ida.liu.se

19th March 2007

1 Introduction

When the paper by Arora *et al.* was published in 1998 traditional analysis of thread scheduling was not multiprogrammed. A set, $\mathcal{P}$, processors was considered to be dedicated. And the task of the thread scheduler was to map threads to the processors with the goal to achieve P-fold speedup.

In addition to dedicated environments this paper studies multiprogrammed environments, i.e., the set of processors is not fixed and may vary while the program is executing. The number of processors available to a certain application is not controlled by the application itself, but by the kernel level scheduler.

The execution of an application is controlled by two levels of schedulers, a user level scheduler and a kernel level scheduler, see Figure 1.

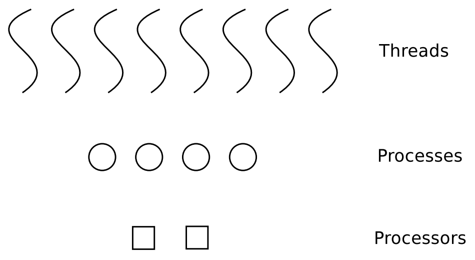

Figure 1: The user level scheduler maps threads to processes while the kernel level scheduler maps processes to a varying set processors.

In the analysis of program execution we consider the processors to execute synchronously in discrete time steps[1].

[*]This was made during the CUGS APP course 2007.

[1]The assumption of synchronous execution is not realistic, but it makes the analysis of

Listing 1: The work stealing algorithm.

```
/* 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();
}
```

We define p_i to be the number of available processors at time step $i \in \mathbb{N}$, and the processor average over T time steps to be

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

Many of the proofs in the paper is based on bounding the execution time, T, by considering

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

2 The work stealing algorithm

The proposed user level scheduler is based on work stealing. One process, called the root process, begins the execution on the initial thread. The rest of the processes, which do not have any work, are thiefs. A thief will try to steal work from a random process, and when it succeeds it will reform and become a regular worker. A sketch of the work stealing algorithm is shown in Listing 1.

The work stealing algorithm relies on every process having a local deque with threads ready to be executed. If a thread spawns a new thread during its execution this new thread (or the running thread) is pushed on the bottom of the local deque. And when a process is blocked or finished executing a thread it pops another thread from the bottom of the local deque and executes that one. When a process steals itpops a thread of the top of another process' deque.

Since more than one process may access a certain deque concurrently there is a need for synchronization. All the dequeus are implemented with *lock-free* synchronization by using a *compare-and-swap* operation supported by hardware.

program execution simpler. The assumption is *not* necessary for the proposed user level scheduler to work.

See Figure 2 for a sketch of the deque. The deque operations are lock-free, and this is cleverly made possible by an additional `tag`-value used together with the cas-operation. The price for having a lock-free implementation is in this case that the `popTop`-operation can fail, that is, not returning a thread to the caller even if there is one available.

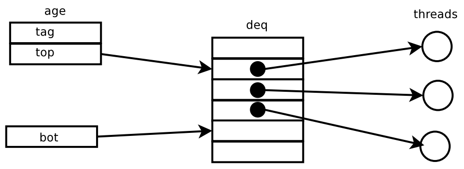

Figure 2: There is a deque on every process. It supports the operations popBottom(), popTop() and pushBottom() (but no pushTop()!).

3 The adversary

The focus in the paper is on the user-level scheduler, and the kernel is viewed as an adversary. There are three kinds of adversaries

- Benign adversary chooses p_i for each time step i and selects processes to execute at random,
- Oblivious adversary chooses (off-line) both p_i and which processes to execute at each time step,
- Adaptive adversary does the same as the oblivious adversary but on-line.

To create good schedules when an adversary makes the kernel schedule requires us to use *yield* system calls. When the adversary is benign we do not need a yield system call to get good expected execution times. In the presence of an oblivious adversary we use a `yieldTo(x)` call that restricts the kernel schedule such that the process that called `yieldTo(x)` will not be executed until process x has executed at least once. The more powerful adaptive adversary requires a more powerful yield, `yieldToAll()`. When a process calls `yieldToAll()` it will not be executed untill all other processes has been executed at least once. Note that the restrictions imposed by the yield calls does not put any restrictions on p_i, only which processes are executed on the available processors.

In the presence of adversaries and yield system calls it is proven that when the work is T_1, the critical path length is T_∞ and the number of processes is P

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

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 $1 - \epsilon$. With this result we see that if we assume that if $P << T_1/T_\infty$ the speedup is linear (asymptoticaly) since $T_\infty P$ is insignificant compared to T_1.