ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Supporting Parallelism in Operating Systems & Programming Languages

Scheduling

Kornilios Kourtis
<kornilios.kourtis@inf.ethz.ch>

Introduction

OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

Challenges/Approaches

Introduction

OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

Challenges/Approaches

Scheduling

mapping of work units to execution units

work units:

- processes
- kernel-level threads
- user-level tasks
- loop iterations

Scheduling

mapping of work units to execution units

work units:

- processes
- kernel-level threads

} OS

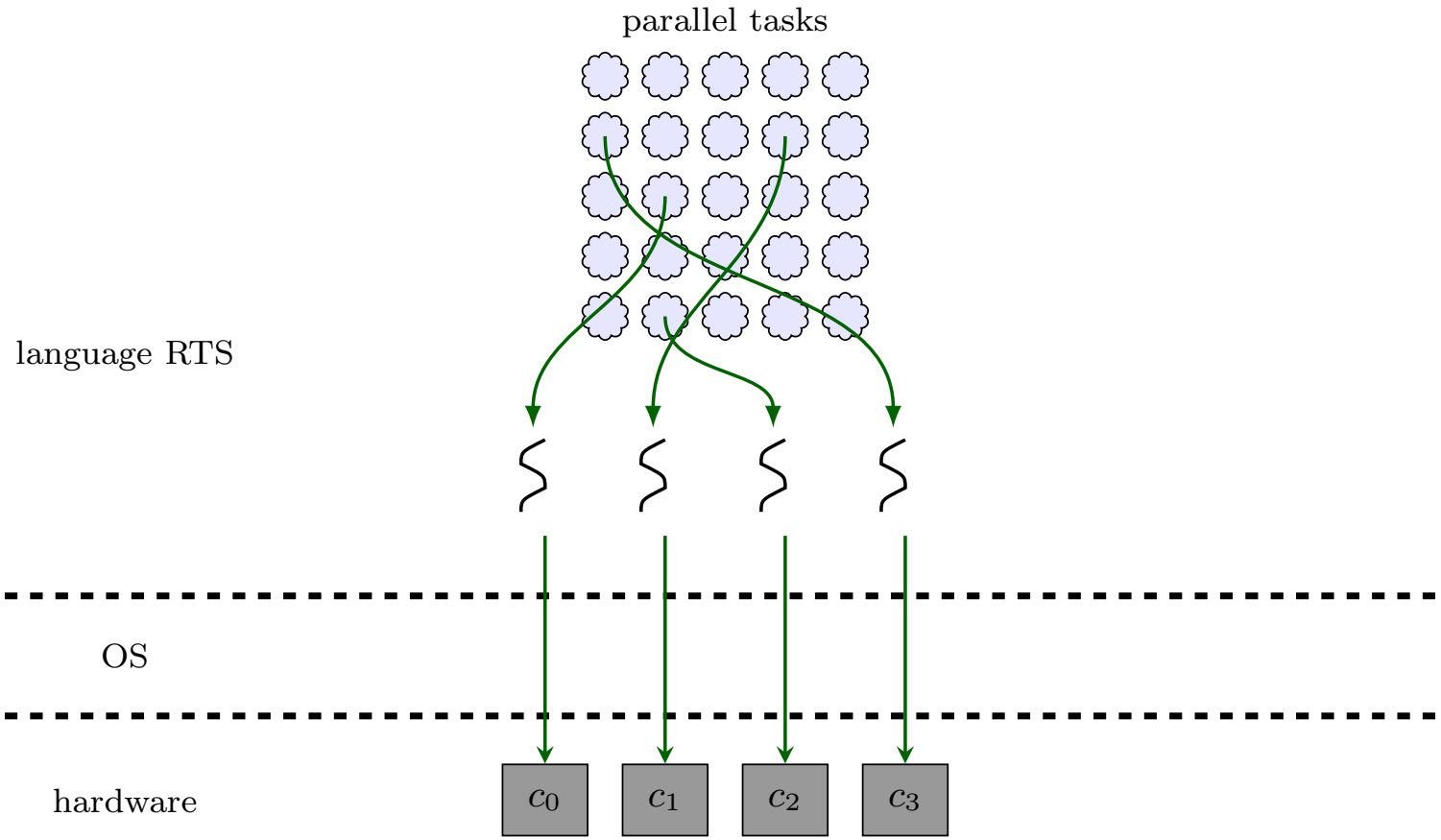
- user-level tasks
- loop iterations

} RTS

two-level scheduling:

- OS (execution unit: cores)
- RTS (execution unit: threads)

Two-level Scheduling



Introduction

OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

Challenges/Approaches

Traditional OS scheduling motivation

- I/O operations
  - take longer than CPU
  - do not need CPU (DMA)
- Time-sharing
  - OS illudes applications that they run exclusively
  - fairness
  - response time

Processes and Threads

- Processes
  - independent address spaces
- (kernel-level) Threads
  - part of a process
  - share address space
  - note: kernel-level threads $\neq$ kernel threads

Next, I will use:

- *process* for kernel-space schedulable entities (includes kernel-level threads)
- *task* for user-space schedulable entities

Preemptive vs Cooperative scheduling

When is the scheduler executed?

Preemptive vs Cooperative scheduling

When is the scheduler executed?

- cooperative scheduling
  - process terminates
  - process blocks (e.g., for I/O)
  - process knows when it will stop

Preemptive vs Cooperative scheduling

When is the scheduler executed?

- cooperative scheduling
  - process terminates
  - process blocks (e.g., for I/O)
  - process knows when it will stop
- preemptive scheduling
  - process becomes runnable (process creation, I/O completed)
  - timer interrupt

Scheduling algorithm

selecting a process to run

- FIFO, SJF (optimal average response time)
- RR (can be unfair to I/O bound processes; why?)
- priority-based, multi-level queues
  - starvation
  - aging
- multi-level feedback queues
  (boost short jobs, I/O bound processes; why?)

What is the scheduling unit?

- kernel-level threads/processes
  - multiple threads $\rightarrow$ more CPU time
- processes (as a collection of kernel-level threads)
- all user processes as one unit

Linux: Scheduling classes

Classes:

- `SCHED_NORMAL`
- `SCHED_BATCH`
- `SCHED_IDLE`
- `SCHED_FIFO`
- `SCHED_RR`

API:

- `sched_setscheduler()`, `sched_getscheduler()`
- `chrt`
- real-time classes generally privileged
  - Real-Time

(old) Linux scheduler

2.6.*x*, $x < 23$

- $\mathcal{O}(1)$ scheduler
- 140 priority queues
  - 0-99: real-time tasks
  - 100-139: user tasks
- dynamic priority
  - boost I/O bound (interactive) tasks
- complex heuristics, difficult to get right
  - audio/video players particularly suffered from this

(new) Linux Scheduler

2.6.x, $x \geq 23$

Hierarchy of schedulers (pluggable)

- real-time
- CFS

Completely Fair Scheduler (CFS)

- tries to give all schedulable entities a fair share of the CPU (respecting priorities)
- uses a red-black tree to store tasks ($\mathcal{O}(logN)$)
- group scheduling
  - tty-based group scheduling (e.g., `make -j8, mplayer`)
- inspired by SD/RSDL fair scheduler

Introduction

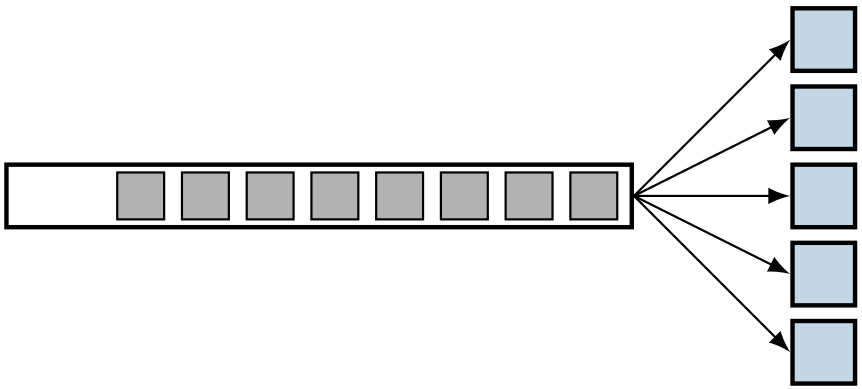
OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

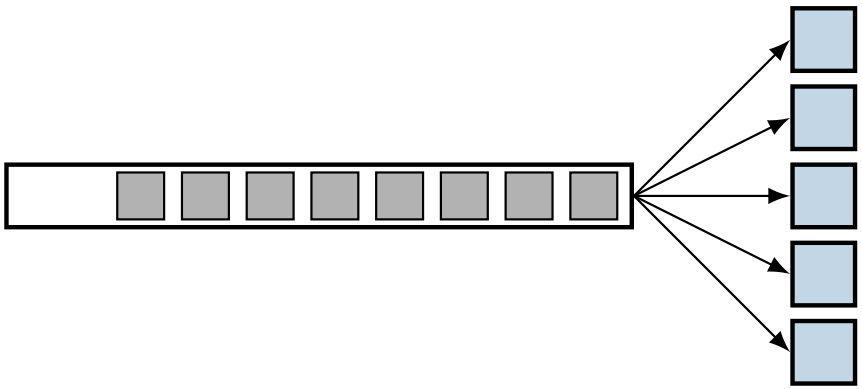
Challenges/Approaches

Single queue



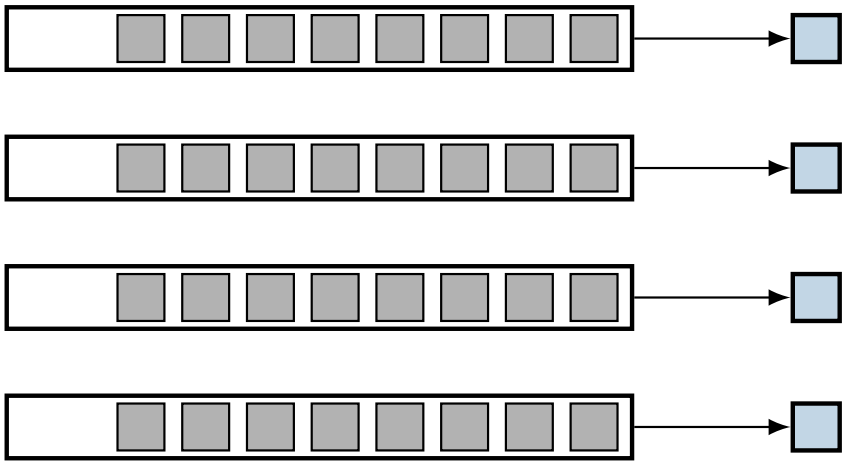
- single task queue for all CPUs
- Issues: ?

Single queue



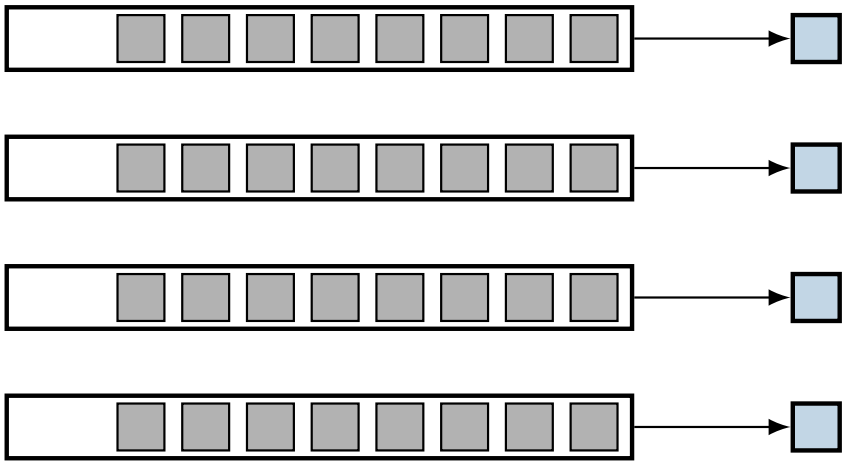
- single task queue for all CPUs
- Issues:
  - contention
  - poor cache-locality

multiple (one-per-cpu) queues



- generally preferred approach
- Issues:?

multiple (one-per-cpu) queues



- generally preferred approach
- Issues:
  - requires rebalancing
  - full utilization not (necessarily) enough
    - tasks in long queues get delayed

Load balancing vs Affinity

Moving things around:

- cache issues
- NUMA issues

Not moving things around:

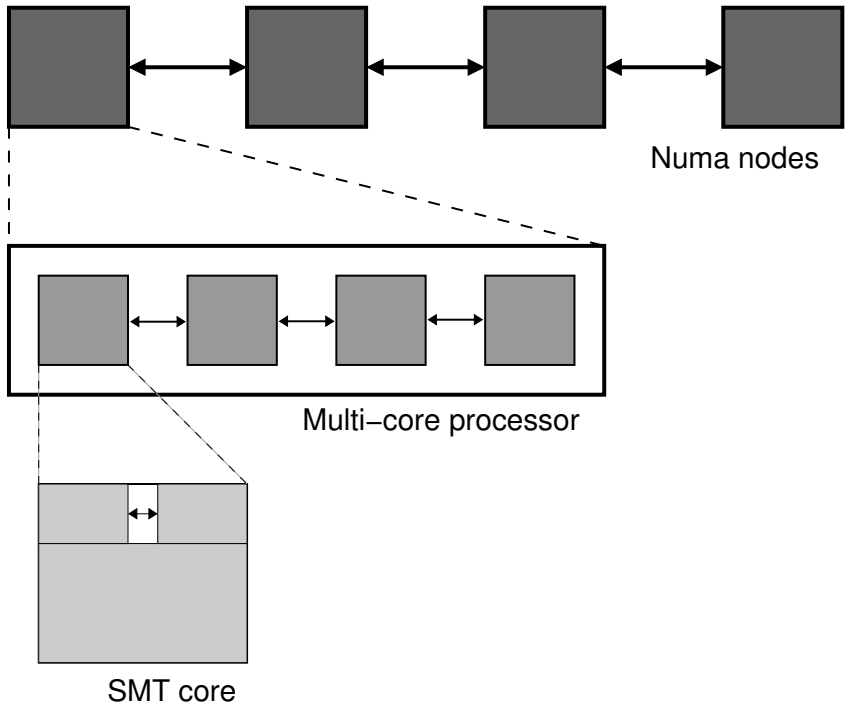
- cores may become idle

Moving a process

- between cores that share a cache
- between cores of different NUMA domains

Linux: Scheduling Domains

- hierarchical structure
- start rebalancing from bottom and move up
- more difficult to move processes across top levels



Linux: scheduling system calls/libraries

- affinity/etc.
  - `sched_setaffinity()`
  - `sched_getaffinity()`
  - `sched_getcpu()`
  - `sched_yield()`
- cpusets
  - affinity on steroids
  - use control groups (cgroups) infrastructure
  - maps group processes to CPUs and memory nodes
  - `libcpuset`

Introduction

OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

Challenges/Approaches

pthreads

(low-level interface)

- POSIX threads API
  - create / synchronize / manage threads
- linux: NPTL library (part of GNU libc)
  - ”1-1”: one kernel-level thread per thread
  - Uses `clone()` syscall
  - Uses `futex()` syscall (synchronization;why?)
- other implementations:
  - ”N-M”: scheduler activations (more on that later!)
  - ”N-1”: Gnu pth (single kernel-level thread)

OpenMP parallel loops

- thread pools
- loop iterations are divided into chunks
  (loop type? data-structure?)
- scheduling policies:
  - static: assigned RR based on thread id
  - dynamic: threads request chunks dynamically
  - guided: chunk $= \frac{\text{remaining iterations}}{\text{threads}}$
- `OMP_DYNAMIC`

Cilk recap

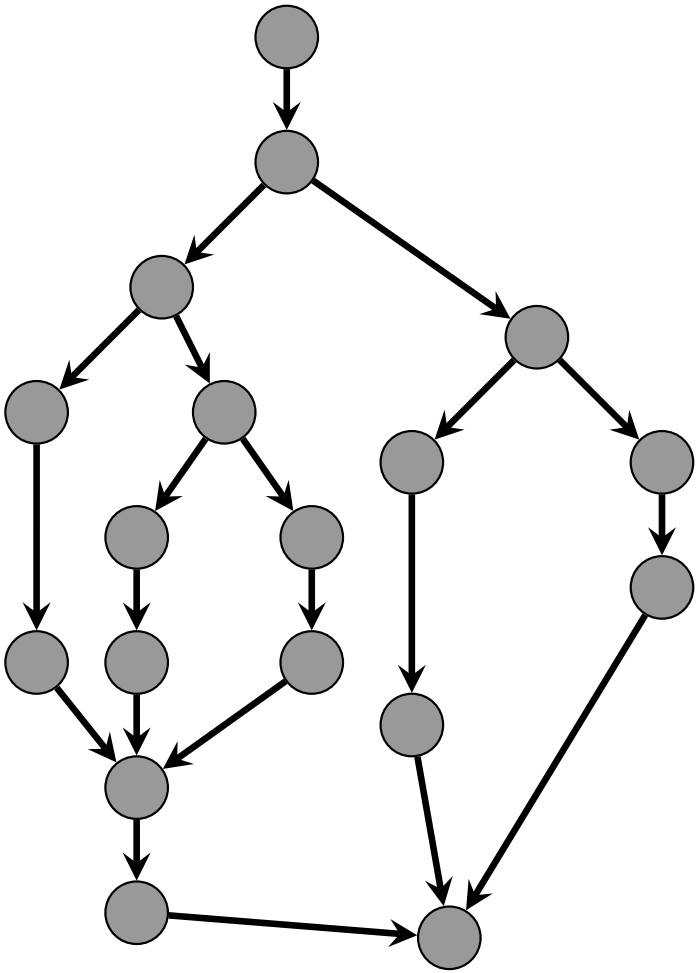
Parallel constructs:

- `spawn`
- `sync`

Why Cilk?

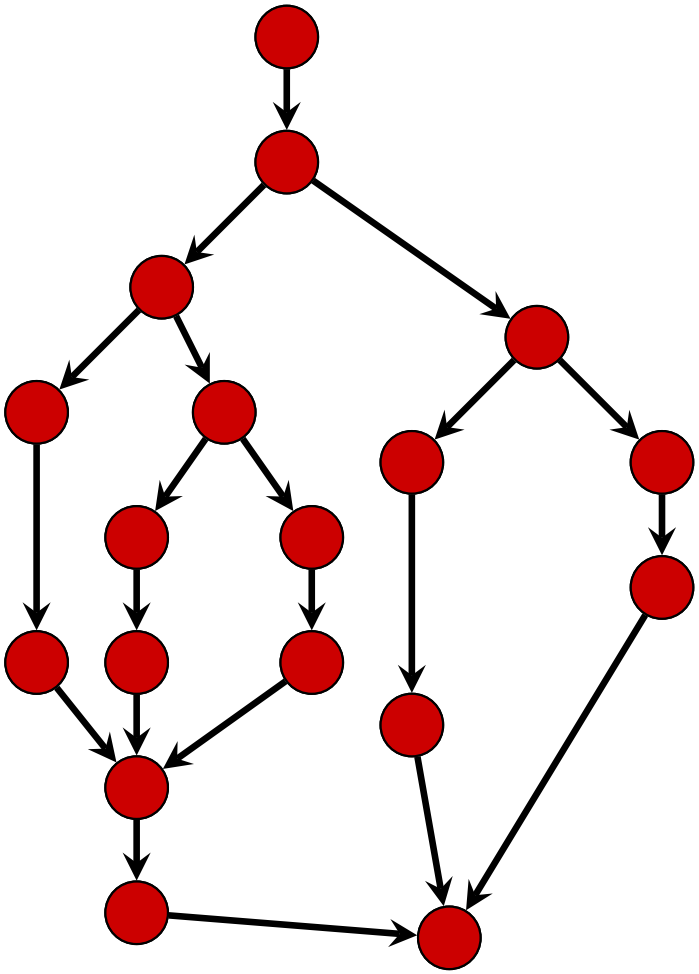
- well-understood, well-documented
- other languages follow similar approach

Cilk Performance model



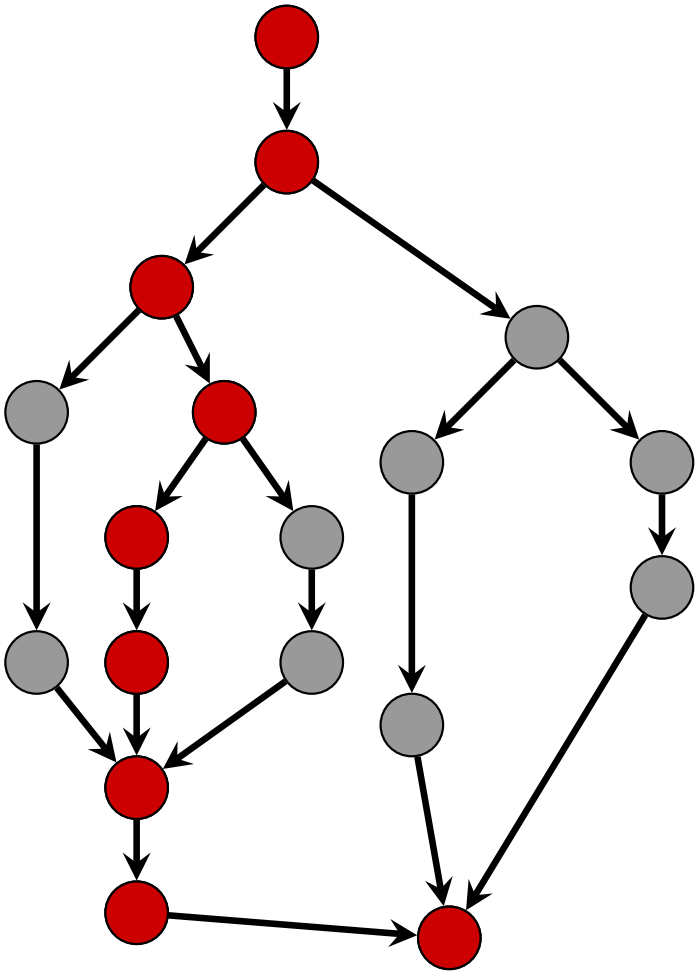
- T_p: Execution time on P CPUs

Cilk Performance model



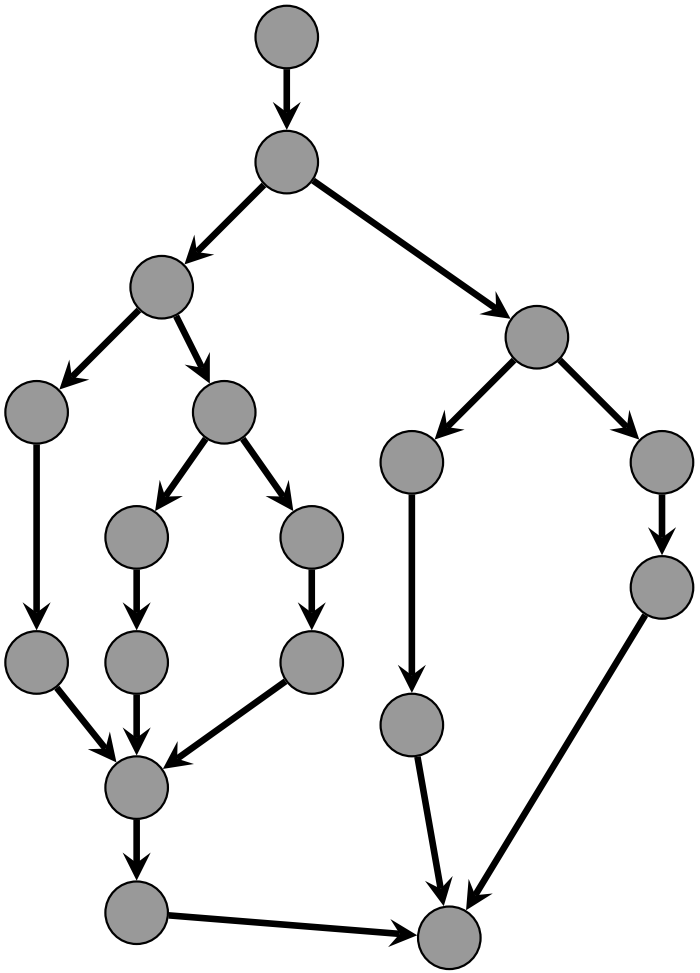
- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)

Cilk Performance model



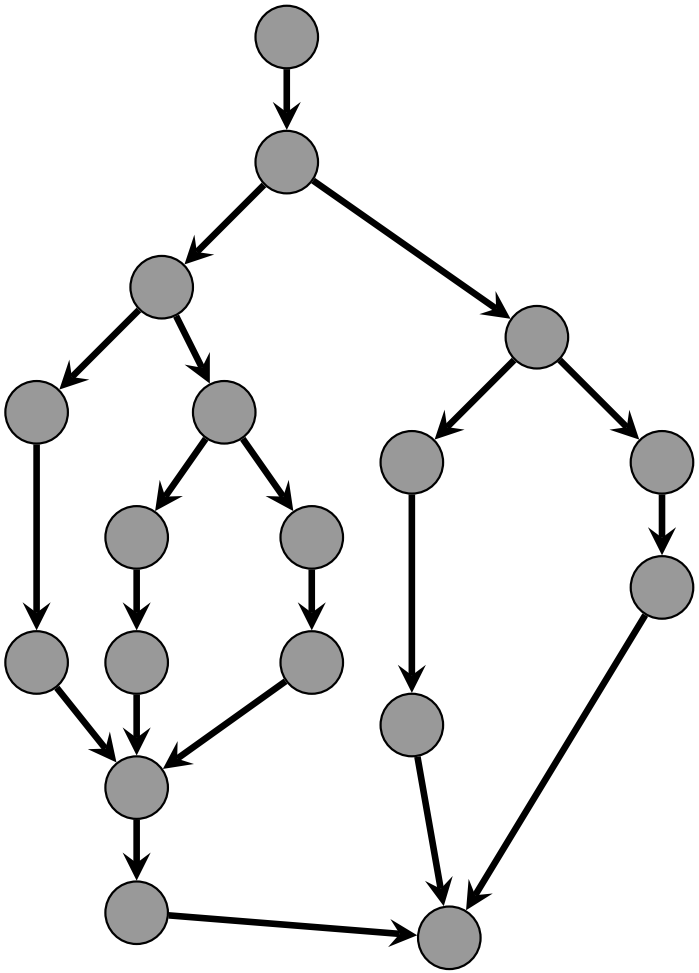
- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)
- T_∞: **span**
  (Execution time for ∞ CPUs)

Cilk Performance model



- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)
- T_∞: **span**
  (Execution time for ∞ CPUs)
- lower bounds: $T_p \geq T_1/P$, $T_p \geq T_\infty$
- maximum speedup: T_1/T_∞

Cilk Performance model



- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)
- T_∞: **span**
  (Execution time for ∞ CPUs)
- lower bounds: $T_p \geq T_1/P$, $T_p \geq T_\infty$
- maximum speedup: T_1/T_∞
- Cilk's scheduler
  - achieves $T_p = T_1/P + \mathcal{O}(T_\infty)$
    (linear speedup if $P \ll T_1/T_\infty$)
  - requires $S_p \leq PS_1$ stack space
  - performs good in practice

Parallel slack

- $P \ll T_1/T_\infty$
- parallel slack: $(T_1/T_\infty)/P$
  - program parallelism T_1/T_∞
  - machine parallelism P

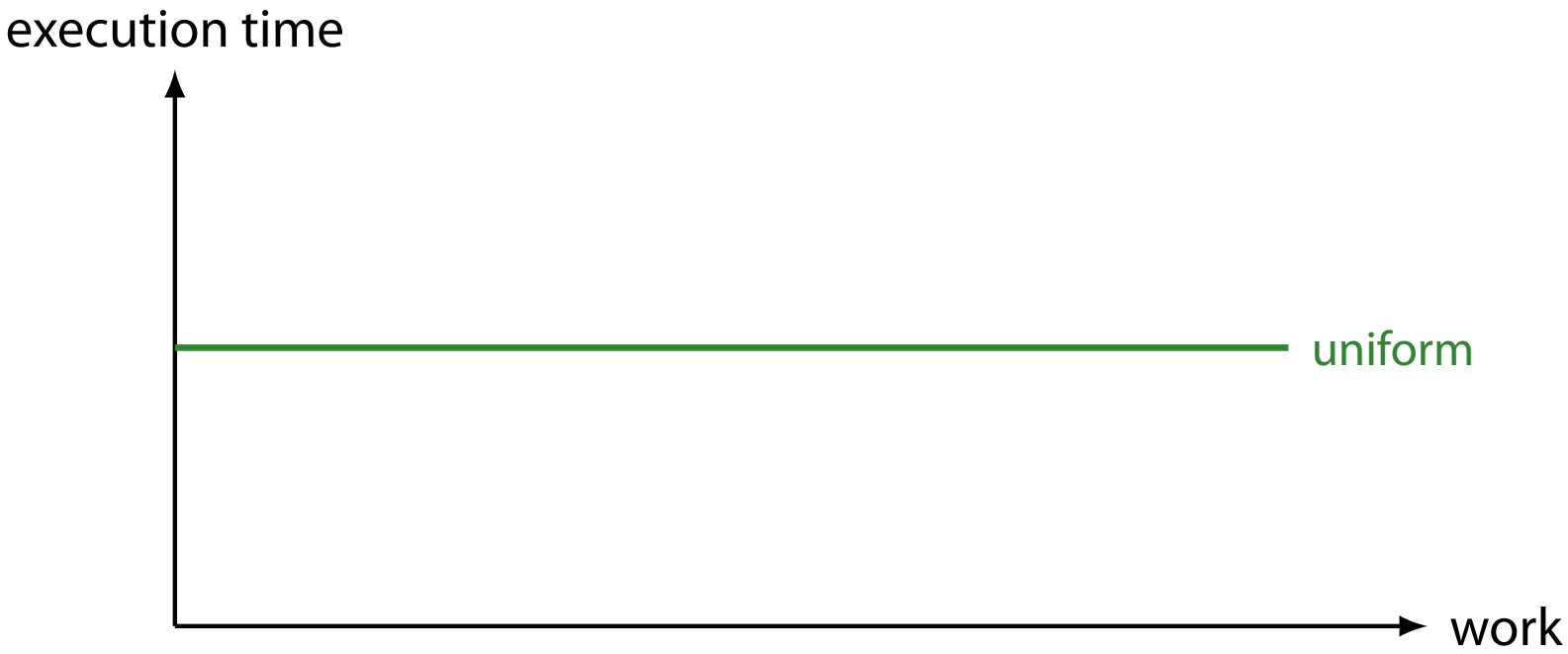
- We need order(s) of magnitute bigger program parallelism than machine parallelism

  Intiuitively:

  + Program does not depend on number of CPUs
  + allows better load balancing

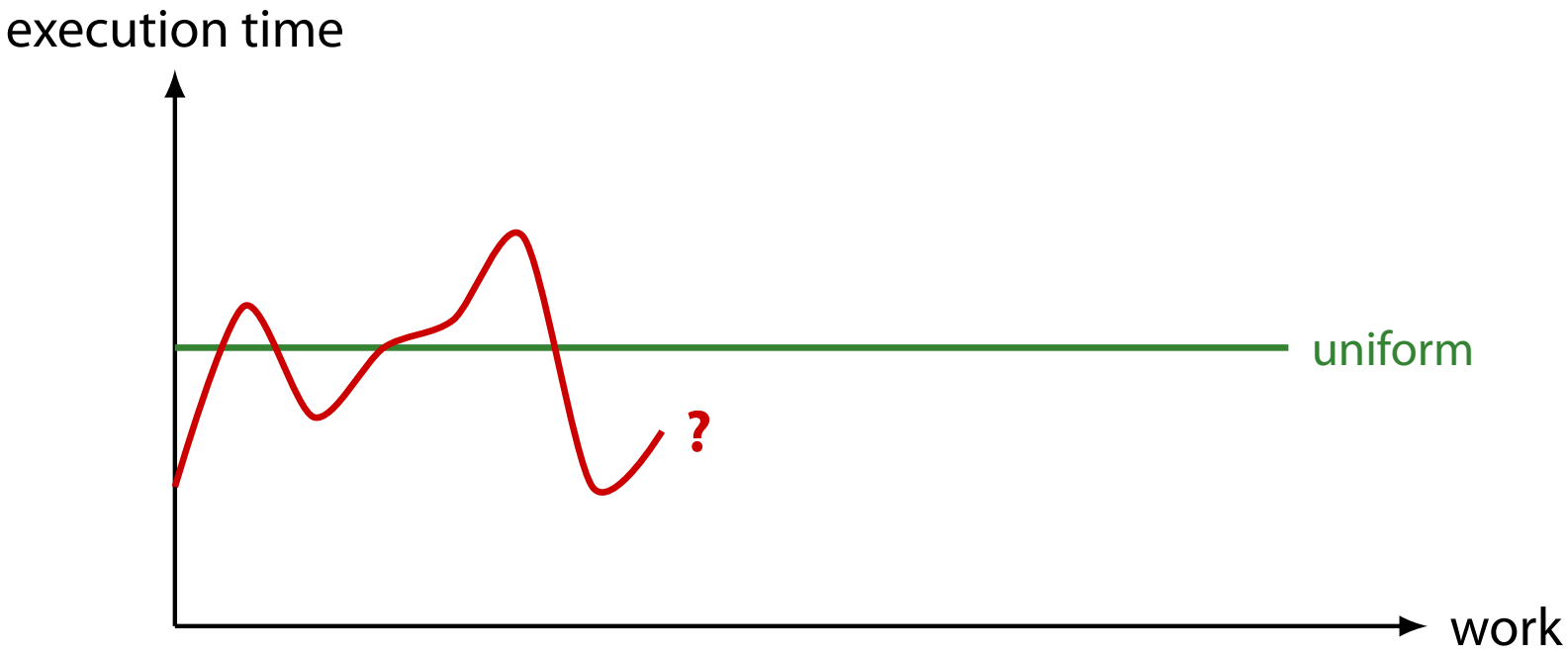
  - **but:**
  - If work for each task small, overhead becomes a significant factor

load balancing - task grain



- Uniform execution time:
  - split work in (equal) #CPU partitions

load balancing - task grain



- Uniform execution time:
  - split work in (equal) #CPU partitions
- Unknown execution time
  - parallel slack provides scheduling flexibility

User-level scheduling for Cilk

- *T* Taks
- *P* kernel-level threads (virtual CPUs)
- tasks can:
  - create other tasks
  - wait for children task to complete

Goals

- load balancing
- efficient space usage
- Small overhead (independent of *T*!)

- When is the scheduler executed?

User-level scheduling for Cilk

- *T* Taks
- *P* kernel-level threads (virtual CPUs)
- tasks can:
  - create other tasks
  - wait for children task to complete

Goals

- load balancing
- efficient space usage
- Small overhead (independent of *T*!)

- When is the scheduler executed?
  - Compiler inserts call in spawn/sync/etc

Scheduling Approaches

- **work sharing:** when new tasks are created, scheduler tries to send them to inactive (virtual) CPUs

Scheduling Approaches

- **work sharing:** when new tasks are created, scheduler tries to send them to inactive (virtual) CPUs
- **work stealing:** Idle processors try to steal tasks
  - child executes first
  - tasks are stolen from the bottom
  - intuitively similar to DFS (good space properties)

work stealing is generally preferred

- better locality
- small synchronization overhead
- optimal theoretic bounds for time/space
  [Blumofe and Leiserson '99]
- **but:** depends on parallel slackness

Scheduling Approaches

- **work sharing:** when new tasks are created, scheduler tries to send them to inactive (virtual) CPUs
- **work stealing:** Idle processors try to steal tasks
  - child executes first
  - tasks are stolen from the bottom
  - intuitively similar to DFS (good space properties)
  - where do we steal from?

work stealing is generally preferred

- better locality
- small synchronization overhead
- optimal theoretic bounds for time/space
  [Blumofe and Leiserson '99]
- **but:** depends on parallel slackness

Scheduling Approaches

- **work sharing:** when new tasks are created, scheduler tries to send them to inactive (virtual) CPUs
- **work stealing:** Idle processors try to steal tasks
  - child executes first
  - tasks are stolen from the bottom
  - intuitively similar to DFS (good space properties)
  - where do we steal from? **random!**

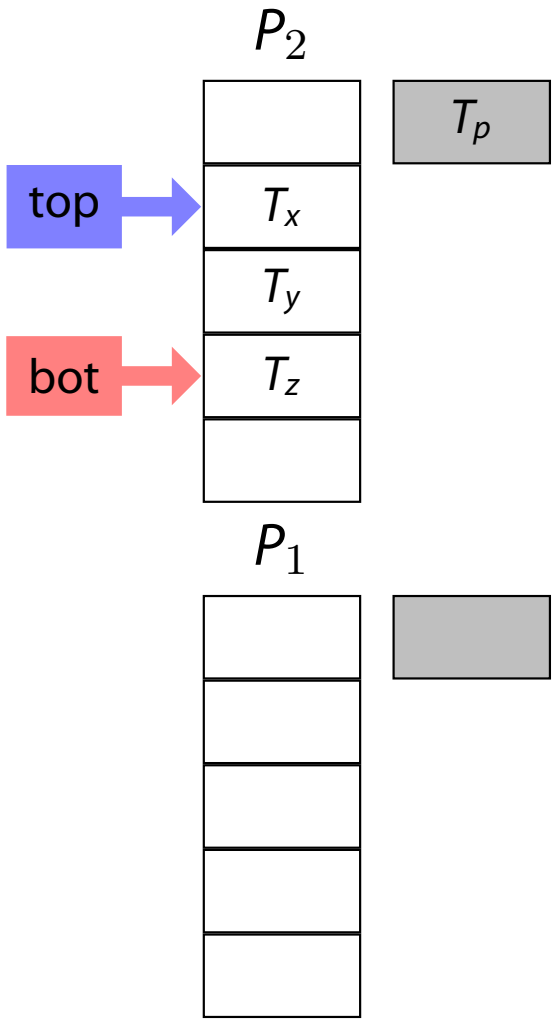
work stealing is generally preferred

- better locality
- small synchronization overhead
- optimal theoretic bounds for time/space
  [Blumofe and Leiserson '99]
- **but:** depends on parallel slackness

work stealing

Cilk Scheduling

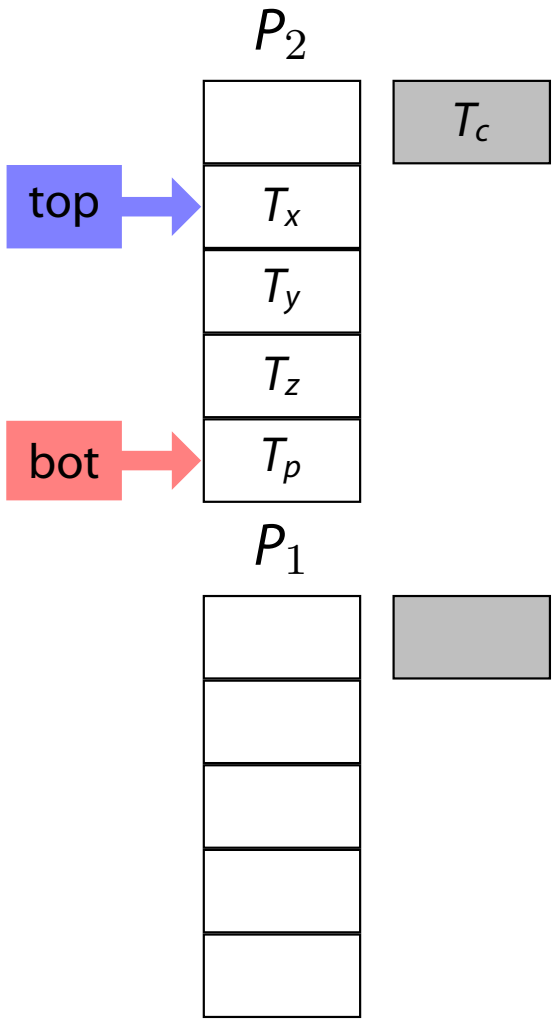
- One scheduler queue per CPU *deque* (double-ended queue, using *THE*)
  - `pushBot`
  - `popBot`
  - `popTop`



work stealing

Cilk Scheduling

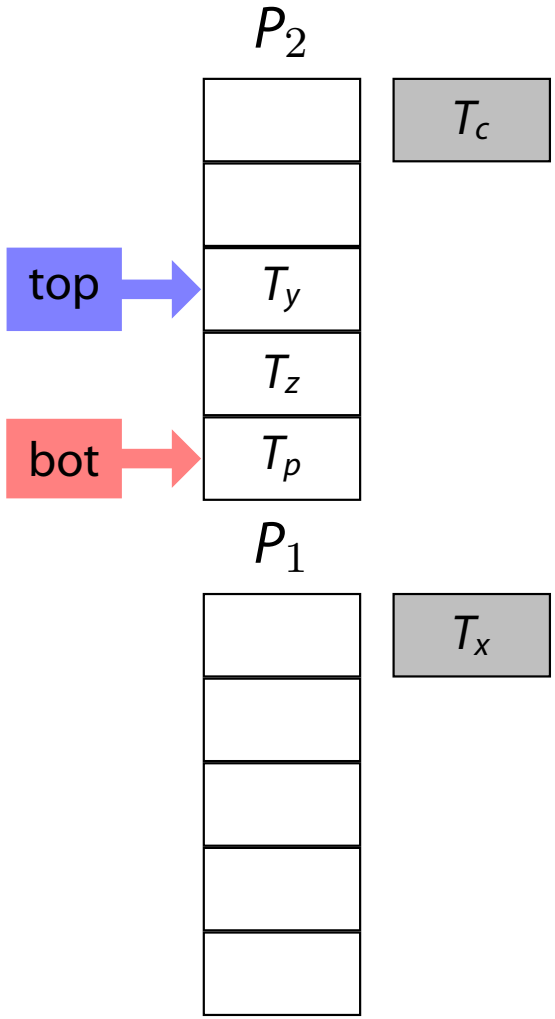
- One scheduler queue per CPU *deque* (double-ended queue, using *THE*)
  - `pushBot`
  - `popBot`
  - `popTop`
- $T_p \rightarrow$ spawn task T_c
  - `pushBot`(T_p)
  - `execute`(T_c) (FAST!)



work stealing

Cilk Scheduling

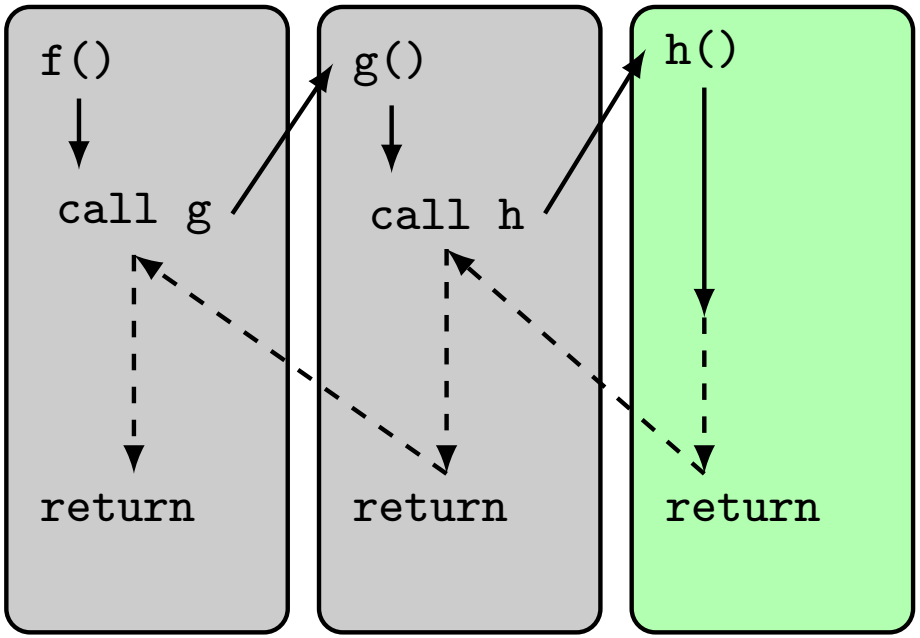
- One scheduler queue per CPU *deque* (double-ended queue, using *THE*)
  - `pushBot`
  - `popBot`
  - `popTop`
- $T_p \rightarrow$ spawn task T_c
  - `pushBot(`T_p`)`
  - `execute(`T_c`)` (FAST!)
- P_1 idle
  - **random** CPU selection p
  - p`->popTop()`
  - execution of stolen task (SLOW!)

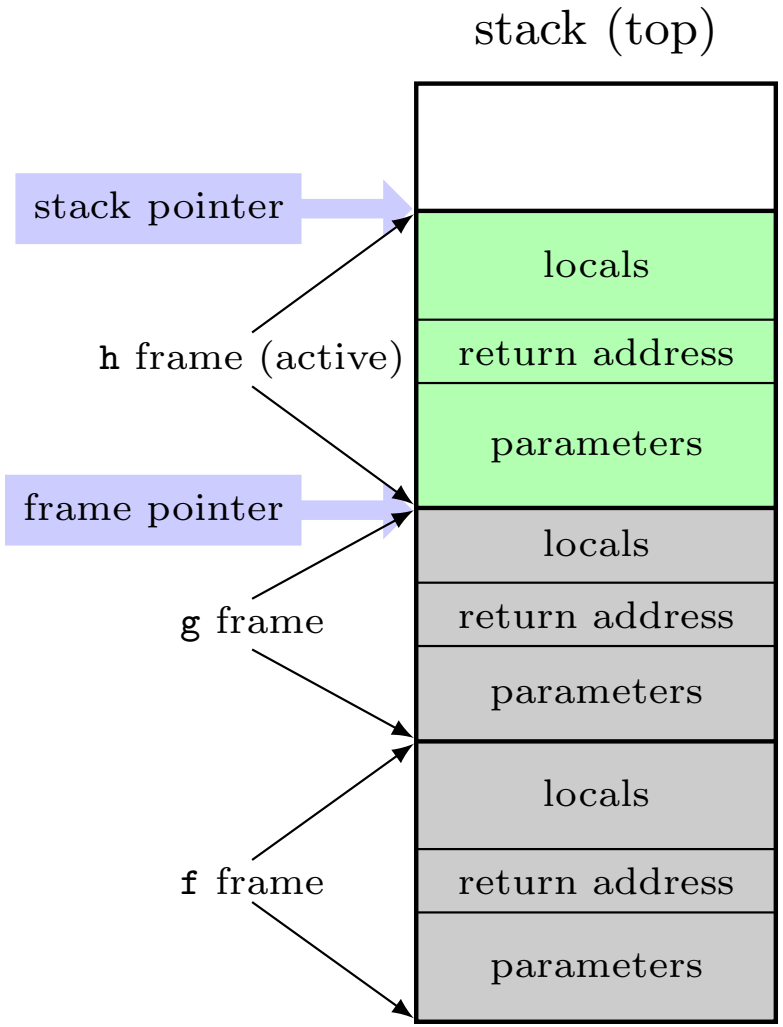


Implementing user-level scheduling

- stealing requirement: each task should be able to run on different processors
- What does a task need to be executed?
  - code position
  - data:
    - Stack
    - Heap
  - registers
- what about kernel-level threads?

recap: execution stack





User-level context switching: non-local exits

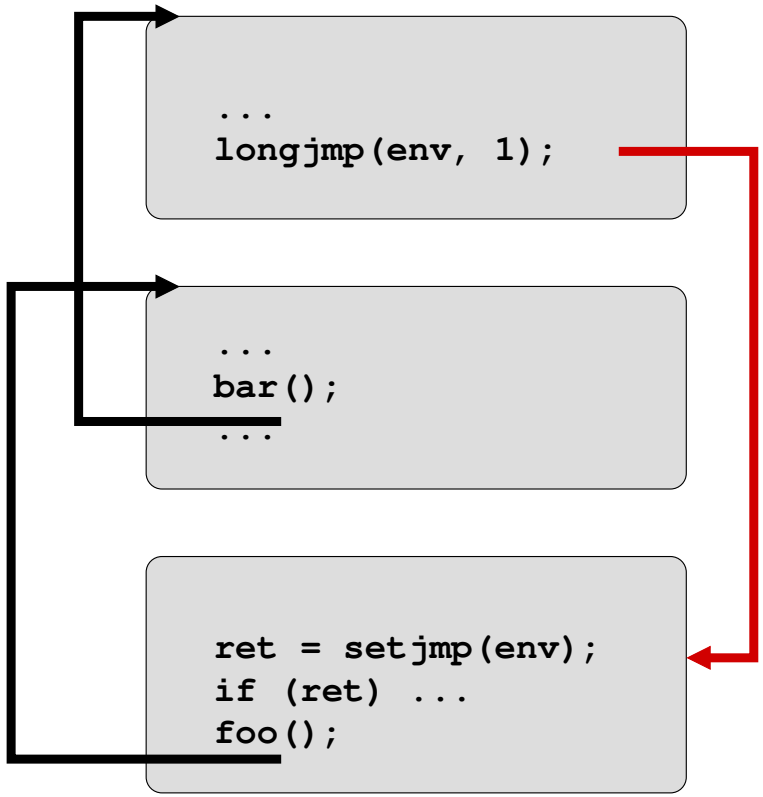
API:

- `int setjmp(jmp_buf);`
- `void longjmp(jmp_buf, int);`

Questions:

- what does `jmp_buf` contain?
- can we use it for context switching?

- more info: GNU libc docs



User-level context switching: ucontext

API:

- ucontext_t
  - .uc_link
  - .uc_sigmask
  - .uc_stack
  - .uc_mcontext
- Functions:

```
int getcontext(ucontext_t *);
// argc *integer* arguments are passed in f
void makecontext(ucontext_t *,
                 void (*f) (void), int argc, ...);
int setcontext(const ucontext_t *);
int swapcontext(ucontext_t *, const ucontext_t *);
```

ucontext example

global definitions:

```
unsigned long count = ...;
ucontext_t us[N];
```

main:

```
ucontext_t u0;
for (unsigned i=0; i<N; i++) {
  ucontext_t *u = us + i;
  getcontext(u);
  u->uc_link = &u0;
  alloc_stack(&u->uc_stack);
  makecontext(
   u, (void (*) (void))f, 1, i);
}
swapcontext(&u0, &us[0]);
```

f:

```
int next_id = (self_id + 1) % N;
ucontext_t *self = us + self_id;
ucontext_t *next = us + next_id;
volatile unsigned long *cnt = &count;
while (--*cnt > 0)
   swapcontext(self, next);
```

ucontext example

global definitions:

```
unsigned long count = ...;
ucontext_t us[N];
```

main:

```
ucontext_t u0;
for (unsigned i=0; i<N; i++) {
  ucontext_t *u = us + i;
  getcontext(u);
  u->uc_link = &u0;
  alloc_stack(&u->uc_stack);
  makecontext(
   u, (void (*) (void))f, 1, i);
}
swapcontext(&u0, &us[0]);
```

f:

```
int next_id = (self_id + 1) % N;
ucontext_t *self = us + self_id;
ucontext_t *next = us + next_id;
volatile unsigned long *cnt = &count;
while (--*cnt > 0)
   swapcontext(self, next);
```

- Passing pointers?

useful ucontext hack

for GNU libc

You can pass pointers on the function used in `makecontext()` in x86_64.

The fine print:

"On architectures where int and pointer types are the same size (e.g., x86-32, where both types are 32 bits), you may be able to get away with passing pointers as arguments to makecontext() following argc. However, doing this is not guaranteed to be portable, is undefined according to the standards, and won't work on architectures where pointers are larger than ints. Nevertheless, starting with version 2.8, glibc makes some changes to makecontext(), to permit this on some 64-bit architectures (e.g., x86-64)."

–makecontext(3)

How cilk does it?

- cilk implements/maintains its own (cactus) stack

  (more on that when we talk about MM)
- Cilk
  - cilk2c compiler
  - variable-based approach; task state is:
    - label
    - live variables
  - cooperative scheduling
    - tasks are not preempted
    - tasks can save/restore its state on their own
- Cilk++ on gcc
  - `__builtin_setjmp()`, `__builtin_longjmp()`

Introduction

OS Scheduling recap

OS Scheduling on multiprocessors

RTS Scheduling

Challenges/Approaches

Challenges

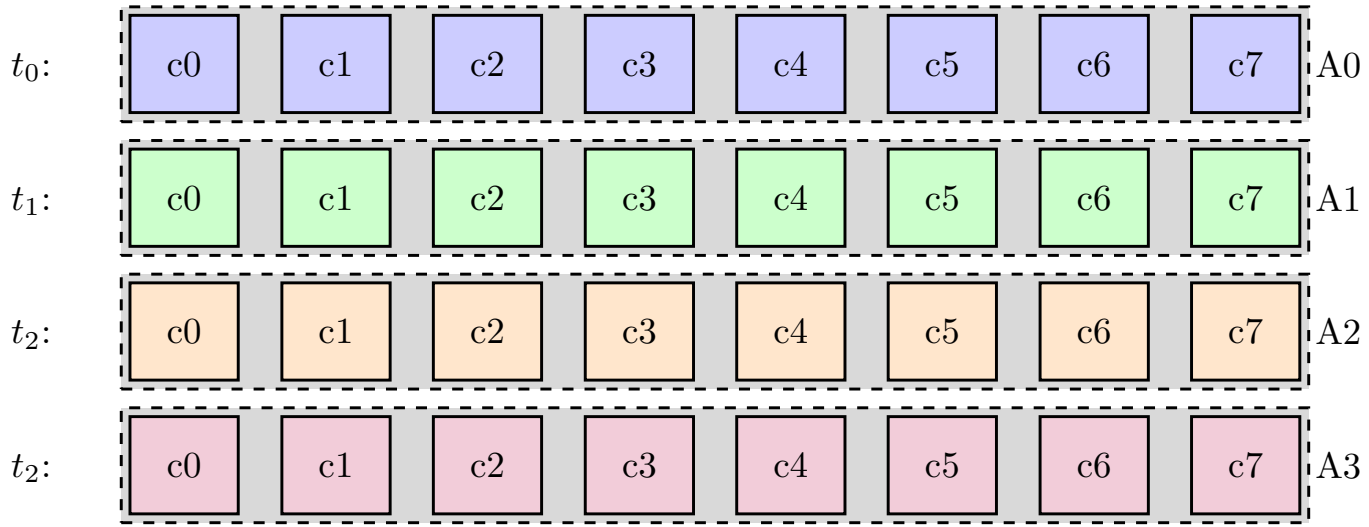
what works (relatively well):

- scheduling sequential applications on multi-processors (OS)
- running parallel applications on dedicated resources (RTS)

Open problem:

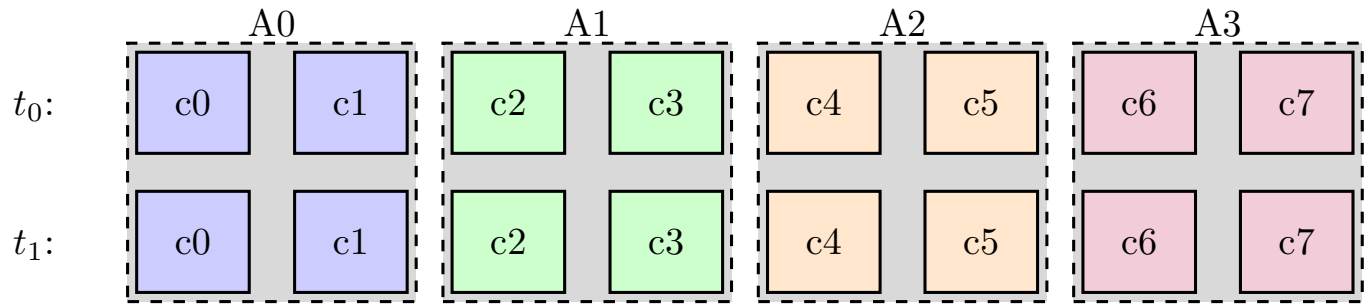
- scheduling parallel applications on multi-processors

Gang scheduling



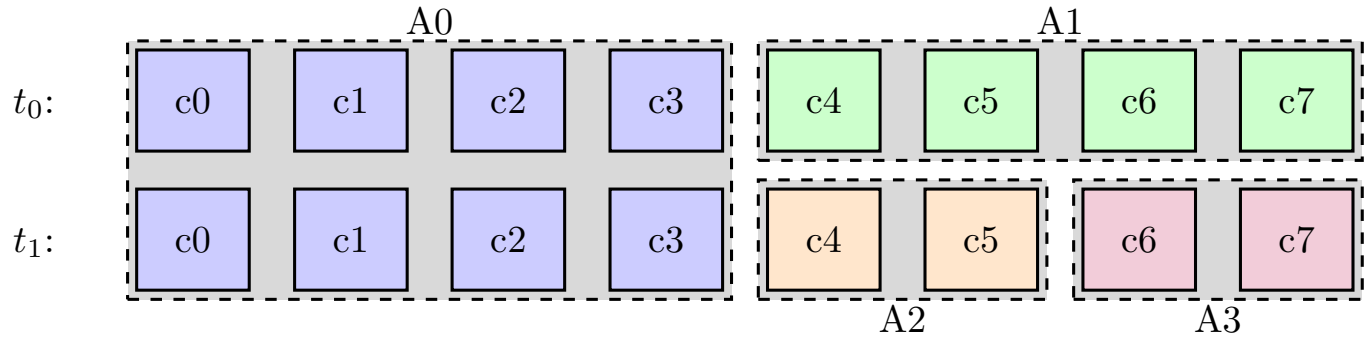
- technique from parallel computing (e.g., clusters)
- synchronization
- potentially bad utilization

space partitioning



- good locality
- fos [Wentzlaff and Agarwal '09]
- does not scale with number of processes

scheduling in time+space



- interface
- policies
- interactive workloads
- [Peter et al. '09]

Scheduling Activations

- [Anderson et al. '92]
- OS support for managing user-level threads
- RTS knows how many (and which) processors are allocated to it
- OS manages parallel applications
- OS $\rightarrow$ RTS
  - #CPUS assigned changed
  - thread blocked (e.g., for I/O)
- RTS $\rightarrow$ OS
  - request/release CPUs

another challenge: heterogeneity

big.LITTLE:

- same ISA
- Cortex-A7 (energy efficient)
- Cortex-A15 (performance)

Scheduling on Linux

- assymetric!
- move from one cluster to another
- switch per-core
- ?

References

- [Blumofe and Leiserson '99] Scheduling multithreaded computations by work stealing (pdf)
- [Frigo et al. '98] The implementation of the Cilk-5 multithreaded language (pdf)
- [Wentzlaff and Agarwal '09] Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores (pdf)
- [Peter et al. '09] Design Principles for End-to-End Multicore Schedulers (pdf)
- [Anderson et al. '92] Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism (pdf)