Parallel Programming with Threads and Tasks

TDDD56 Lecture 2

Christoph Kessler

PELAB / IDA
Linköping university
Sweden

2014

Outline

Lecture 1: Multicore Architecture Concepts

- Architectural trends and consequences for programming

Lecture 2: Parallel programming with threads and tasks

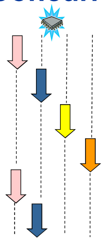
- Revisiting processes, threads, synchronization
  - Pthreads
  - OpenMP (very shortly)
- Tasks
  - Cilk
  - Futures

Lecture 3: Shared memory architecture concepts

Lecture 4: Non-blocking synchronization

Lecture 5: Design and analysis of parallel algorithms

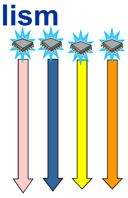
Concurrency vs. Parallelism



Concurrent computing

1 or few CPUs

Quasi-simultaneous execution



Parallel computing

Many CPUs

Simultaneous execution of many / all threads of the *same application*

Common issues:
- threads/processes for overlapping execution
- synchronization, communication
- resource contention, races, deadlocks

Goals of concurrent execution:
- Increase CPU utilization
- Increase responsitivity of a system
- Support multiple users

Central issues: Scheduling, priorities, …

Goals of parallel execution:
- Speedup of 1 application (large problem)

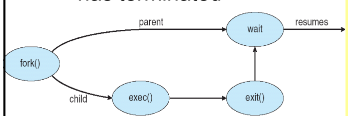
Central issues: Parallel algorithms and data structures, Mapping, Load balancing…

Processes

(Refresher from TDDB68)

Example: Process Creation in UNIX

- **fork** system call
  - creates new child process
- **exec** system call
  - used after a **fork** to replace the process' memory space with a new program
- **wait** system call
  - by parent, suspends parent execution until child process has terminated



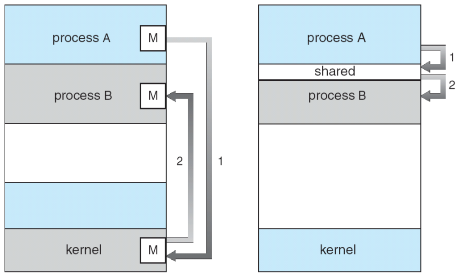
C program forking a separate process

```
int main()
{
   Pid_t ret;
   /* fork another process: */
   ret = fork();
   if (ret < 0) { /* error occurred */
        fprintf ( stderr, "Fork Failed“ );
        exit(-1);
   }
   else if (ret == 0) { /* child process */
        execlp ( "/bin/ls", "ls", NULL );
   }
   else { /*parent process: ret=childPID *
        /* will wait for child to complete: *
        wait (NULL);
        printf ("Child Complete");
        exit(0);
   }
}
```

Parallel programming with processes

- Processes can create new processes that execute concurrently with the parent process
- OS scheduler – also for single-core CPUs
- Different processes share nothing by default
  - Inter-process communication via OS only, via shared memory (write/read) or message passing (send/recv)
- **Threads** are a more light-weight alternative for programming shared-memory applications
  - Sharing memory (except local stack) by default
  - Lower overhead for creation and scheduling/dispatch
    - E.g. Solaris: creation 30x, switching 5x faster

IPC Models – Realization by OS



(a)

IPC via **Message Passing**

(b)

IPC via **Shared Memory**

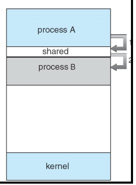
Syscalls: send, recv

Syscalls: shmget, shmat, then load / store

Example: POSIX Shared Memory API

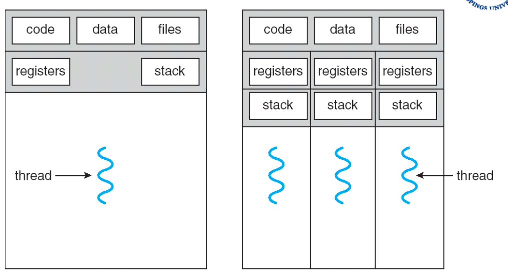
- #include <sys/shm.h>
  #include <sys/stat.h>

- Let OS create a shared memory segment (system call):
  - int segment_id = **shmget** (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
- Attach the segment to the executing process (system call):
  - void *shmemptr = **shmat** (segment_id, NULL, 0);
- Now access it:
  - strcpy ((char *)shmemptr, "Hello world"); // Example: copy a string into it
  - …
- Detach it from executing process when no longer accessed:
  - **shmdt** (shmemptr);
- Let OS delete it when no longer used:
  - **shmctl** (segment_id, IPC_RMID, NULL);



Threads

Single- and Multithreaded Processes



single-threaded process

multithreaded process

A **thread** is a basic unit of CPU utilization:

- Thread ID, program counter, register set, stack.

A process may have one or several threads.

Benefits of Multithreading

- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
- **Utilization of Multiprocessor Architectures**
  - Convenient (but low-level) shared memory programming

POSIX Threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread programming in C
  - start and terminate threads
  - coordinate threads
  - regulate access to shared data structures
- API specifies behavior, not implementation, of the thread library
- C interface, e.g.
  - **int** pthread_create (pthread_t *thread, **const** pthread_attr_t *attr, **void** *(*start_routine)(**void***), **void** *arg);
- Note: as a library, rely on underlying OS and hardware!
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Starting a Thread (1)

- Thread is started with function

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

  - Called func must have parameter and ret values void*
    - Exception: first thread is started with main()
  - Thread terminates when called function terminates or by pthread_exit (**void** *retval)
  - Threads started one by one
  - Threads represented by data structure of type pthread_t

Starting a Thread (2)

- Example:

```
#include <pthread.h>

int main ( int argc, char *argv[] )
{
  int *ptr;
  pthread_t thr;

  pthread_create( &thr,
                  NULL,
                  foo,
                  (void*)ptr  );
  …
  pthread_join( &thr, NULL );
  return 0;
}
```

```
void *foo ( void *vp )
{
   int  i  =  (int) vp;;
   …
}


// alternative
//  – pass a parameter block:
void *foo ( void *vp )
{
   Userdefinedstructtype *ptr;
   ptr=(Userdefinedstructtype*)vp;
   …
}
```


Access to Shared Data (0)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 (foo1) could even write to thread2's (foo2) stack frame

- **Example 0: Parallel incrementing**

```
int a[N];    // shared,   assume P | N
pthread_t  thr[P];

int main( void )
{
   int t;
   for (t=0; t<P; t++)
      pthread_create(&(thr[t]), NULL,
                     incr, a + t*N/P );
   for (t=0; t<P; t++)
      pthread_join( thr[t], NULL );
    …
}
void *incr ( void *myptr_a )
{  int i;
   for (i=0; i<N/P; i++)
       ((int*)myptr_a[i])++;    }
```


Access to Shared Data (1)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 (foo1) could even write to thread2's (foo2) stack frame

- **Example 1**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;
   globalptr = &i;  // ??? dangerous!
     // if foo1 terminates, foo2 writes
     // somewhere, unless globalptr
     // value is reset to NULL manually
     …
}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```


Access to Shared Data (2)

- Globally defined variables are globally shared and visible to all threads
- Locally defined variables are visible to the thread executing the function
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 could even write to thread2's stack frame

- **Example 2**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;
   globalptr =(int*)malloc(sizeof(int));
   // safe, but possibly memory leak;
   // OK if garbage collection ok


}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```


Coordinating Shared Access (3)

What if several threads need to write a shared variable?

- If they simply write: ok if write order does not play a role
- If they read and write: encapsulate (critical section, monitor) and protect e.g. by mutual exclusion using mutex locks)
- **Example: Access to a taskpool**
  - threads maintain list of tasks to be performed
  - if thread is idle, gets a task and performs it

```
// each thread:
while (! workdone)
{
   task = gettask( Pooldescr );
   performtask ( task );
}

// may be called concurrently:
Tasktype gettask ( Pool p )
{
   // begin critical section
   task = p.queue [ p.index ];
   p.index++;
   // end critical section

   return task;
}
```

Race Conditions lead to Nondeterminism

- Example: p.index++
- could be implemented in machine code as

```
39: register1 = p.index            // load
40: register1 = register1 + 1      // add
41: p.index = register1            // store
```

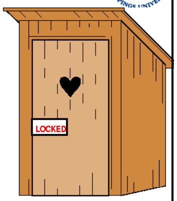
Not atomic!

- Consider this execution interleaving, with “index = 5” initially:

```
39: thread1 executes register1 = p.index                { T1.register1 = 5 }
39: thread2 executes register1 = p.index                { T2.register1 = 5 }
40: thread1 executes register1 = register1 + 1          { T1.register1 = 6 }
40: thread2 executes register1 = register1 + 1          { T2.register1 = 6 }
41: thread1 executes p.index = register1                { p.index = 6 }
41: thread2 executes p.index = register1                { p.index = 6 }
```

- Compare to a different interleaving, *e.g.*, 39,40,41, 39,40,41…
  → Result depends on relative speed of the accessing threads *(race condition)*

Critical Section



- **Critical Section**: A set of instructions, operating on shared data or resources, that should be executed by a single thread at a time without interruption
  - **Atomicity** of execution
  - **Mutual exclusion**: At most one process should be allowed to operate inside at any time
  - **Consistency**: inconsistent intermediate states of shared data not visible to other processes outside
- May consist of different program parts for different threads
  - that access the same shared data
- General structure, with structured control flow:

  ...
  Entry of critical section C
  … critical section C: operation on shared data
  Exit of critical section C

Coordinating Shared Access (4)

```
pthread_mutex_t mutex;    // global variable definition - shared
   …
// in main:
   pthread_mutex_init( &mutex, NULL );
   …

// in gettask:
   …
   pthread_mutex_lock( &mutex );
   task = p.queue [p.index];
   p.index++;
   pthread_mutex_unlock( &mutex );
   …
```

Often implemented using test_and_set or other atomic instruction where available

Hardware Support for Synchronization

- Most systems provide hardware support for protecting critical sections
- Uniprocessors – could ***disable interrupts***
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special ***atomic instructions***
  - **TestAndSet**: test memory word and set value atomically
    - Atomic = non-interruptable
    - If multiple TestAndSet instructions are executed *simultaneously* (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.
  - **AtomicSwap**: swap contents of two memory words atomically
  - **CompareAndSwap**
  - **Load-linked / Store-conditional**

TestAndSet Instruction

- Definition in pseudocode:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;      // return the OLD value
}
```

atomic

Mutual Exclusion using TestAndSet

- Shared boolean variable lock, initialized to FALSE (= unlocked)
-

```
do {
    while ( TestAndSet (&lock ))
        ;   // do nothing but spinning on the lock (busy waiting)
    // … critical section

    lock = FALSE;
    // … remainder section

} while ( TRUE);
```

→EXAMPLE

Compare-And-Swap

see Lecture 4

- CompareAndSwap (adr_memcell, value, register)
  - Atomically compares a value in a memory cell to a supplied value and, if these are equal, swaps the contents of the memory cell with the value stored in a register.
- **Example: Mutual Exclusion using Compare-And-Swap:**

```
register = 1;
CompareAndSwap ( &lock, 0, register );
while (register != 0)
        CompareAndSwap ( &lock, 0, register );
// … critical section
lock = 0;
//
```

- More details in the lecture on non-blocking synchronization

Load-linked / Store-conditional

see Lecture 4

2 new instructions for memory access:

- **LoadLinked** *address, register*
  - records the version number of the value read (cf. a svn update)
- **StoreConditional** *register, address*
  - will only succeed if no other operations were executed on the accessed memory location since my last LoadLinked instruction to *address*, (cf. a svn commit)
  - and set the register operand of Store-conditional to 0, otherwise.

Mutual Exclusion using LL/SC

see Lecture 4

- Shared int variable lock, initialized to 0 (= unlocked)
- **do** {

```
    register = 0;
    while ( register == 0) {
        dummy = LoadLinked ( &lock );
        if (dummy == 0) {   // read a 0 – found unlocked
            register = 1;
            register = StoreConditional ( register, &lock );
        }
        // if register is 0, StoreConditional failed, retry…
    }
    // … critical section
    lock = 0;   // ordinary store

    // …   remainder section
} while ( TRUE);
```

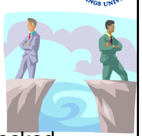

Pitfalls with Semaphores

- Correct use of mutex operations:
  - Protect all possible entries/exits of control flow into/from critical section:

    pthread_mutex_lock (&mutex)
    ….
    pthread_mutex_unlock (&mutex)

- Possible sources of synchronization errors:
  - Omitting **lock**(&mutex) or **unlock**(&mutex) (or both) ??
  - **lock**(&mutex) …. **lock**(&mutex) ??
  - **lock**(&mutex1) …. **unlock**(&mutex2) ??
  - if-statement in critical section, **unlock** in then-branch only

Problems: Deadlock and Starvation



- **Deadlock** – two or more threads are waiting indefinitely for an event that can be caused only by one of the waiting threads
  - Typical example: *Nested critical sections*
    - Guarded by locks S and Q, initialized to unlocked

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
…	…
signal (S);	signal (Q);
signal (Q);	signal (S);

time

- **Starvation** – indefinite blocking. A thread may never get the chance to acquire a lock if the mutex mechanism is not *fair*.

Deadlock Characterization

[Coffman *et al.* 1971]

Deadlock can arise only if **four conditions** hold simultaneously:

- **Mutual exclusion:** only one thread at a time can use a resource.
- **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads.
- **No preemption of resources:** a resource can be released only voluntarily by the thread holding it, *after* that thread has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, …, P_n\}$ of waiting threads such that
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for a resource that is held by P_2, …,
  - P_{n-1} is waiting for a resource that is held by P_n, and
  - P_n is waiting for a resource that is held by P_0.

Coordinating Shared Access (5)

- Must also rely on implementation for efficiency
- Time to lock / unlock mutex or synchronize threads varies widely between different platforms
- A mutex that all threads access serializes the threads!
  - Convoying
  - Goal: Make critical section as short as possible

```
// in gettask():
int tmpindex;  // local (thread-private) variable
pthread_mutex_lock( &mutex );
tmpindex = p.index++;
pthread_mutex_unlock( &mutex );
task = p.queue [ tmpindex ];
```

Possibly slow shared memory access now outside critical section

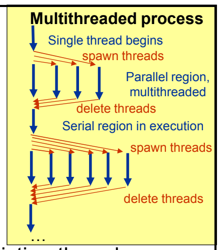
Coordinating Shared Access (6)

- When programming on this level of abstraction: can minimize serialization, but not avoid
  - Example: Fine-grained locking
- Better: avoid mutex and similar constructs, and use higher-level data structures that are lock-free
  - Example: NOBLE
- Also: Transactional memory

More about this in Lecture 4

Performance Issues with Threads on Multicores

Performance Issue: Thread Pools



- For a multithreaded process: Create a number of threads in a pool where they await work
- Advantages:
  - Faster to service a request with an existing thread than to create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Win32 API
- OpenMP

Performance Issue: Spinlocks on Multiprocessors

- Recall busy waiting at spinlocks:

```
// … lock initially 0 (unlocked)
while ( ! test_and_set( &lock ))
    ;
// … the critical section …
lock = 0;
```

- Test_and_set in a tight loop → high bus traffic on multiprocessor
  - Cache coherence mechanism must broadcast all writing accesses (incl. t&s) to lock immediately to all writing processors, to maintain a consistent view of lock's value
  - → contention
  - → degrades performance

Solution 1: TTAS

- Combine with ordinary read:

```
while ( ! test_and_set( &lock ))
    while ( lock )
        ;
// … the critical section …
```

- Most accesses to lock are now reads
  → less contention, as long as lock is not released.

Solution 2: Back-Off

- ```
  while ( ! test_and_set( &lock ))
      do_nothing_for ( short_time );
  // … the critical section …
  ```
- Exponential / random back-off

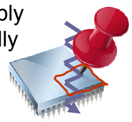
Performance Issue: Manual Avoidance of Idle Waiting

- Thread that unsuccessfully tried to acquire mutex is blocked but not suspended
  - busy waiting, idle ☹
- Can find out that mutex is locked and do something else:
  pthread_mutex_trylock (&mutex_lock);
  - If mutex is unlocked, returns 0
    If mutex is locked, returns EBUSY
- Useful for locks that are not accessed too frequently and for threads having the chance to do something else
  ```

Performance Issue: Thread Pinning

Programmer may want tight control over thread-to-core mapping of the underlying OS CPU scheduler, e.g. due to:

- Caching, NUMA ("non-uniform memory access [time]")
  → not all cores are equally "close" to all memory locations
  → map thread to a specific core (subset) it has "affinity" to
- Application / runtime system might do its own load balancing (see later)
- Measurement reproducibility problem
  - Measured time, energy can vary considerably if the thread mapping changes unexpectedly



Solution: "pin" a thread to a specific core

- Constrain CPU scheduler to always map it to that same core

37

Thread Pinning with Pthreads

- **#include <pthread.h>**

int pthread_setaffinity_np (
pthread_t *thread*,
size_t *cpusetsize*,
const cpu_set_t **cpuset*
);

Bitvector-like data structure describing a subset of the available CPU set): i th bit =1 iff core i is in the affinity set of this thread.

int pthread_getaffinity_np (
pthread_t *thread*,
size_t *cpusetsize*,
cpu_set_t **cpuset*
);

38

Thread Pinning with Pthreads

- Pthreads provides macros for defining and comparing CPU sets.
- Example: pin the current thread to core i (in 0,…,p-1):

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

cpu_set_t cpuset;

CPU_ZERO( &cpuset );      // empty CPU mask
CPU_SET ( i, &cpuset );    // set i-th "bit" in cpuset

pthread_t this_thread = pthread_self ( ); // get this thread's handle

pthread_setaffinity_np ( this_thread, sizeof(cpu_set_t), &cpuset );
```

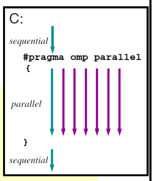
Better Programmability for Thread Programming

Short overview of OpenMP™

(see TDDC78 for in-depth treatment of OpenMP)

OpenMP™

- Standard for shared-memory thread programming
- Developed for incremental parallelization of HPC code
- Directives (e.g. #pragma omp parallel)
- Support in recent C compilers, e.g. gcc from v.4.3 and later
- High-level constructs for data and work sharing
  - Low-level thread programming still possible




```
#include <omp.h>
…
#pragma omp parallel  shared(N)  private(i)
{    // creating a team of OMP_NUM_THREADS threads
   …
#pragma omp for  schedule(static)
   for (i=0;  i<N;  i++)
       domuchwork( i );
}
```

Work (here: iterations of for loop) shared among all threads of the current team

→EXAMPLE

41

Performance Issue: Load Balancing

- Longest-running process determines parallel execution time ("makespan")
- Minimized by **load balancing**
  - Static – mapping of tasks to cores *before* runtime, no OH
  - Dynamic – mapping done *at* runtime
    - Shared (critical section) or distributed work pool
    - On-line problem – don't know the future, only the past
      - Heuristics such as best-fit, random work stealing

Example: Parallel loop, iterations of unknown+varying workload

```
#pragma omp parallel for  schedule(dynamic)
for (i=0;  i<N; i++)   work ( i, unknownworkload(i) );
```

42

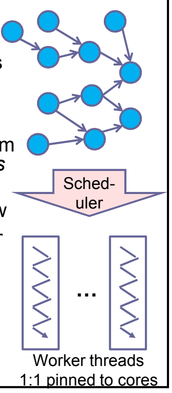
Programming with Tasks

(rather than threads)

Example: Cilk

Task-level programming

- Threads are too inflexible, still too coarse-grained, only moderately portable
- Idea: **Program for tasks, not for threads**
- Lots of tasks, with explicit or implicit dependences
  - Created dynamically by the application
  - Execute non-preemptively
  - Maintained and scheduled by a run-time system running (at user level) on top of *worker threads* provided by underlying hardware/OS
  - Scheduling can be driven by operand data-flow
  - Central task pool vs. Work-stealing scheduler
- Examples:
  - Cilk
  - OpenMP 3.0 task model and runtime system
  - StarPU for heterogeneous / hybrid multicores
  - StarSS / OmpSS / SMPSS



Cilk

- supertech.csail.mit.edu/cilk
- algorithmic multithreaded language
- programmer specifies parallelism and exploits data locality
- runtime system schedules computation to a parallel platform
- extension of C
- fork-join execution style
  - typ. overhead for spawning on SMP ca. 4x time for subroutine call
- Commercial branch Cilk++ for C++, bought by Intel in 2009
  - Intel Cilk™ Plus, part of Intel Parallel Building Blocks

Cilk: Fine-grained multithreading

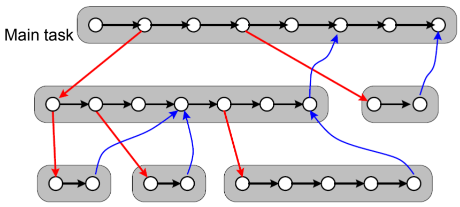
```
cilk int fib (int n)
{
  if (n < 2) return n;
  else {
     int x, y;
     x = spawn fib (n-1);
     y = spawn fib (n-2);
     sync;
     return (x+y);
  }
}
```

- Expose massive, fine-grained thread-level parallelism (*tasks*)
- Restricted synchronization – no mutual exclusion, possibly non-preemptable execution
- Use low-overhead dynamic (work-stealing) task scheduler for automatic load balancing

→ Raise the abstraction level: program for tasks, not threads

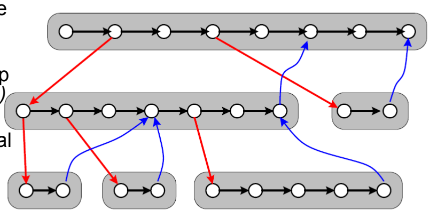
Execution of Cilk programs

- Cilk tasks may execute local tasks (black edges = continuation of control flow in execution trace), spawn child tasks or synchronize with child tasks (blue dependence edges).
  - Execution forms a directed acyclic graph (DAG), task graph
  - DAG depth = Length of longest path (critical path) is a lower bound on parallel execution time



Execution of Cilk Programs (2)

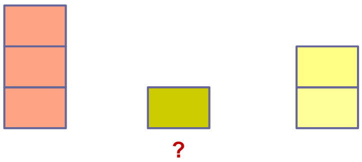
- **Execution schedule**:
  - maps tasks to *processors* X *time steps* (Gantt chart)
  - follows the DAG precedence constraints
  - each processor executes at most one task per time step
- A task is <u>ready for scheduling</u> if all its dependencies are saturated.
- Simple dynamic scheduling algorithm: **Greedy (list) scheduling**
  - Maintain central task pool and the set ("front") of ready tasks
  - At each time step issue all (max. *p*) *ready tasks*
- In practice, decentral work-stealing schedulers work well.



Work-Stealing

- Each processor / core maintains its own local task pool
  - A doubly-ended queue (deque)
  - Spawned tasks (frames) pushed on top
  - Processor pops its next task from top

  (Like an ordinary local call stack → low overhead unless idle)
  - Idle processor steals a task from a randomly chosen victim processor's task queue bottom
    - If the victim is also idle, try another one…

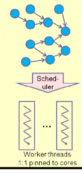


Work-Stealing – Implementation (1)

```
taskqueue  tasks;   // one per worker thread

void spawn ( task t )
{
   myworker.tasks.push( t );
}

void sync() // wait for last spawned subtask:
{
  (status, t) = myworker.tasks.pop();
  if (status == STOLEN)  {
    while (! t.done)
                                 ;    // wait for t
    return t.result;
  }
  else // have to do it myself:
    return t.execute();
}
```




```
void steal_work( victim )
{
  t = victim.tasks.steal();
  if (t != NULL)  {  // got one:
    t.thief = myworker;
    t.result = t.execute();
    t.done = TRUE;
  }
}

thread  worker ( id, root_task )
{
  if (id == 0)
     root_task.execute();
  else
     forever
        steal_work (rand_victim());
}
```

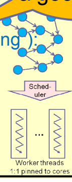
Work-Stealing – Implementation (2)

```
taskqueue  tasks;   // one per worker thread

void spawn ( task t )
{
   myworker.tasks.push( t );
}

void sync() // wait for last spawned subtask:
{
  (status, t) = myworker.tasks.pop();
  if (status == STOLEN)  {
    while (! t.done)
       // try to steal back ("leapfrogging"):
       steal_work( t.thief );
    return t.result;
  }
  else // have to do it myself:
    return t.execute();
}
```

Why is this a good idea?




```
void steal_work( victim )
{
  t = victim.tasks.steal();
  if (t != NULL)  {  // got one:
    t.thief = myworker;
    t.result = t.execute();
    t.done = TRUE;
  }
}

thread  worker ( id, root_task )
{
  if (id == 0)
     root_task.execute();
  else
     forever
        steal_work (rand_victim());
}
```

Futures

Implicit task creation and synchronization with Futures

Futures

Here: in Java (5+) Concurrency

```
class FibTask implements Callable<Integer>
{
  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask( int n )
  {
     arg = n;
  }
  public Integer call()
  {
    if (arg > 2) {
      Future<Integer> left  = exec.submit( new FibTask( arg – 1 ));
      Future<Integer> right = exec.submit( new FibTask( arg – 2 ));
      return  left.get() + right.get();
    } else
      return 1;
  }
}
```

A **Callable**<T> object represents a task that return a value of type T. Method **call**() (without parameter) returns the result.

Creates a **thread pool** to which tasks can be submitted, waited for, and killed.

Java thread-pools abstract from platform-specific thread management

spawn two tasks to compute fib(arg-1) and fib(arg-2), with handles left and right, respectively

get(): blocks until future value (arg) has been written or task is completed

See e.g. chapter 16 [Herlihy/ Shavit'08]

Futures

```
Future<Integer> left  = exec.submit( new FibTask(arg– 1));
… = left.get() + … ;
```

- A ***future call*** by a thread T1 starts a new thread T2 to calculate one or more values and allocates a ***future cell*** for each of them.
- T1 is passed a read-reference to each future cell and continues immediately.
- T2 is passed a write-reference to each future cell
- Such references can be passed on to other threads
- As (T2) computes results, it writes them to their future cells.
- When any thread touches a future cell via a read-reference, the read stalls until the value has been written.
- A future cell is written only once but can be read many times.
- Used e.g. in Tera-C [Callahan/Smith'90], ML+futures [Blelloch/Reid-Miller'97], StackThreads/MP [Taura et al.'99], Java (5+) Concurrency Package [SUN'04], C++11

Questions?

Further Reading (Selection)

- C. Lin, L. Snyder: *Principles of Parallel Programming*. Addison Wesley, 2008. (general introduction; Pthreads)
- B. Wilkinson, M. Allen: *Parallel Programming, 2e*. Prentice Hall, 2005. (general introduction; pthreads, OpenMP, MPI)
- M. Herlihy, N. Shavit: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. (threads; nonblocking synchronization)
- Chandra, Dagum, Kohr, Maydan, McDonald, Menon: *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.
- Barbara Chapman et al.: *Using OpenMP - Portable Shared Memory Parallel Programming*. MIT press, 2007.

Other references

- OpenMP: www.openmp.org
- NOBLE – Library of non-blocking data structures www.cse.chalmers.se/research/group/noble/
- Cilk – algorithmic multithreading supertech.csail.mit.edu/cilk
- StarPU runtime system for heterogeneous multicores runtime.bordeaux.inria.fr/StarPU
- StarSs / OmpSs www.bsc.es