The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

MCS 572 Lecture 11
Introduction to Supercomputing
Jan Verschelde, 20 September 2024

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

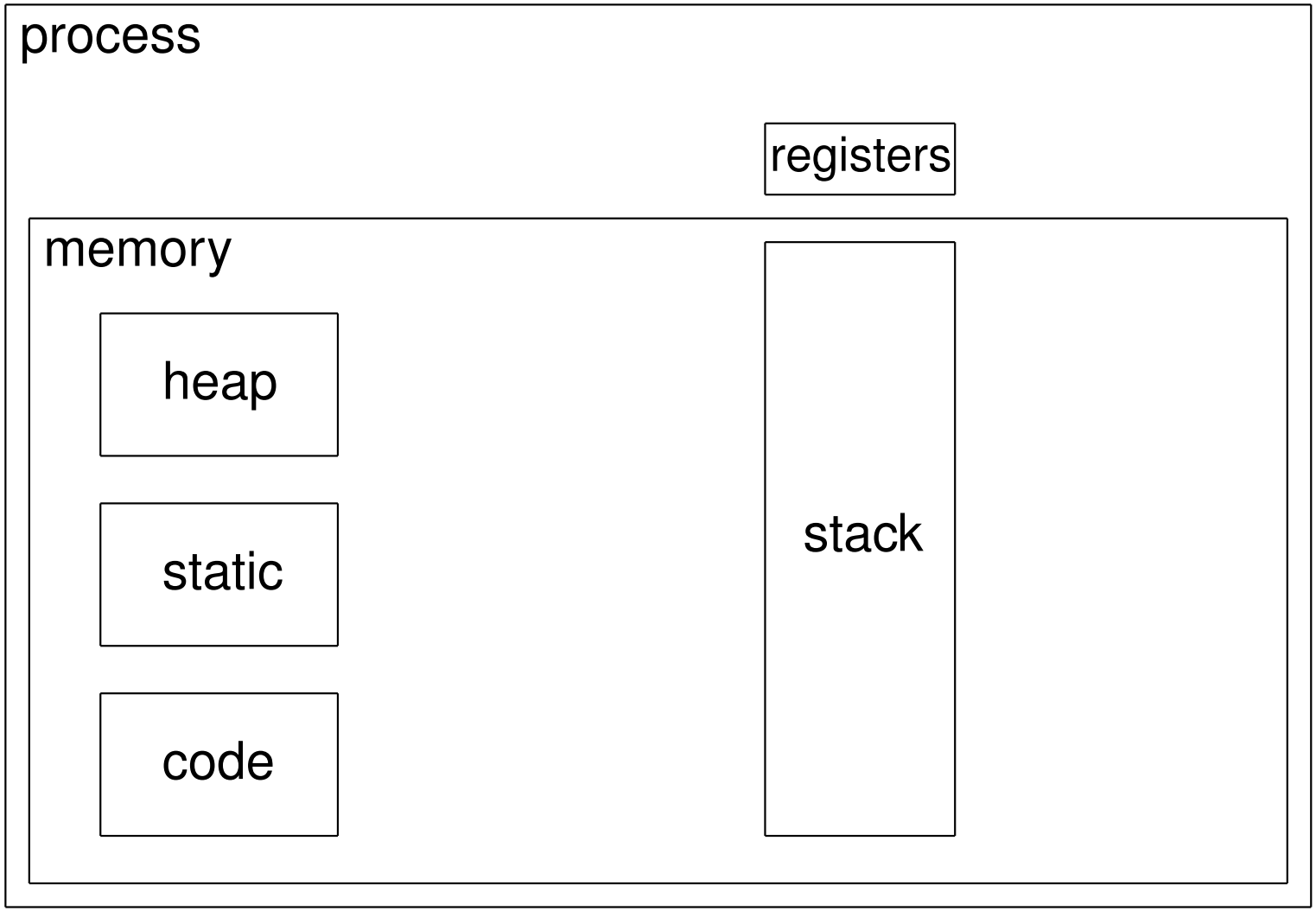
processes and threads

A thread is a single sequential flow within a process.

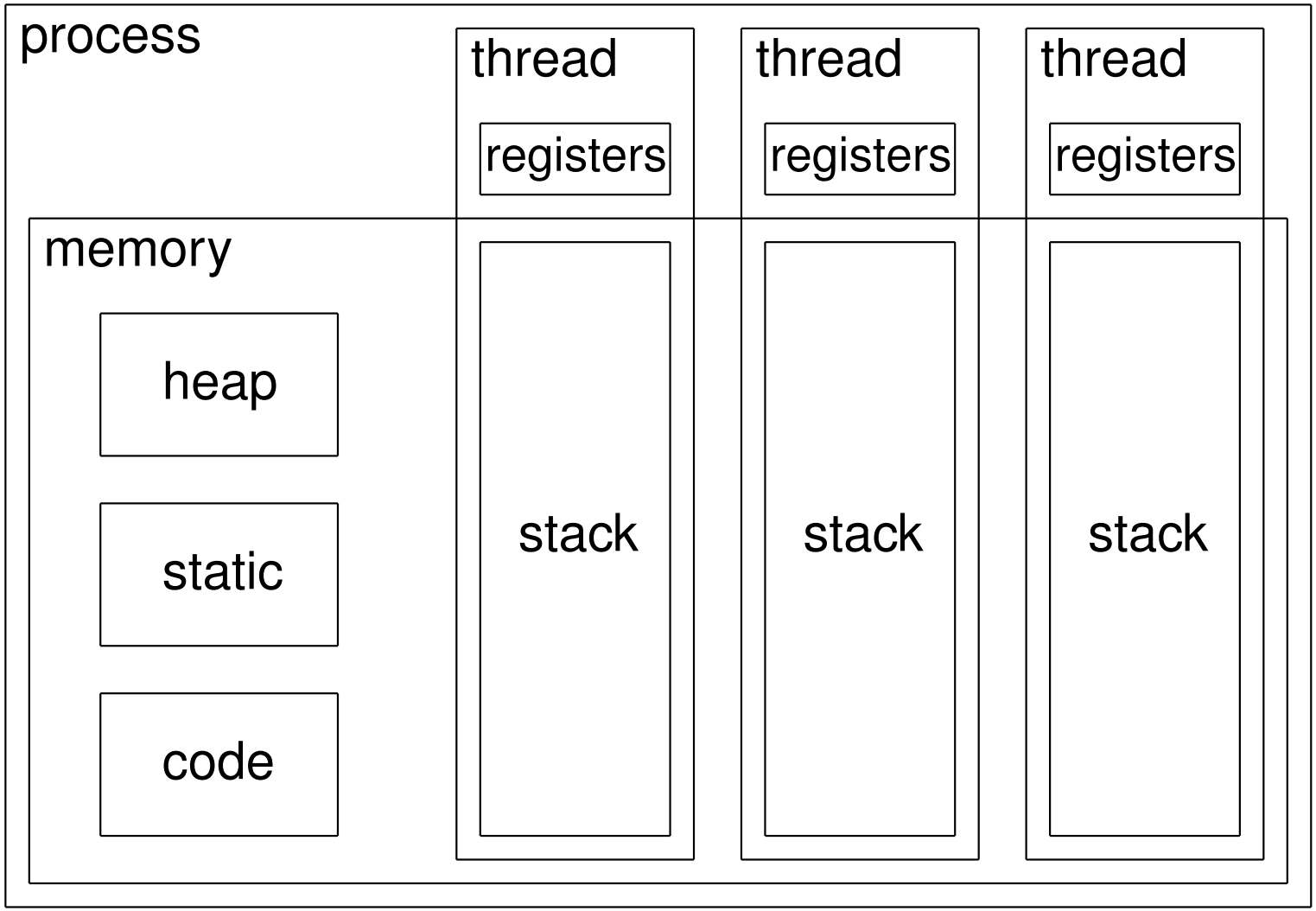
Multiple threads within one process share heap storage,
static storage, and code.
Each thread has its own registers and stack.

Threads share the same single address space and synchronization is needed when threads access same memory locations.

single threaded process



multithreaded process



processes and threads

A thread is a single sequential flow within a process.

Multiple threads within one process share

- heap storage, for dynamic allocation and deallocation,
- static storage, fixed space,
- code.

Each thread has its own registers and stack.

Difference between the stack and the heap:

- stack: Memory is allocated by reserving a block of fixed size on top of the stack. Deallocation is adjusting the pointer to the top.
- heap: Memory can be allocated at any time and of any size.

Threads share the same single address space and synchronization is needed when threads access same memory locations.

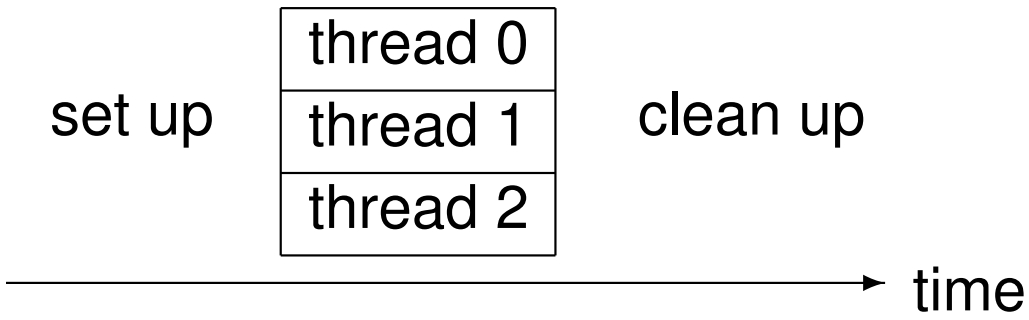
The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

the work crew model

Instead of the manager/worker model where one node is responsible for the distribution of the jobs and the other nodes are workers, with threads we can apply a more collaborative model.

A computation performed by three threads in a work crew model:



If the computation is divided into many jobs stored in a queue, then the threads grab the next job, compute the job, and push the result onto another queue or data structure.

processing a queue of jobs

We will simulate a work crew model:

- Suppose we have a queue with n jobs.
- Each job has a certain work load (computational cost).
- There are p threads working on the n jobs.

To distribute the jobs among the threads:

- either each worker has its own queue of jobs,
- or idle workers do to the next job in the shared queue,
- or idle workers with empty job queues steal jobs.

We will consider the second type of distributing jobs.

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

a crew of workers with Julia

```
% julia -t 3 workcrew.jl
The jobs : [4, 5, 5, 2, 4, 6, 3, 6, 6, 4]
the number of threads : 3
Worker 1 is ready.
Worker 3 is ready.
Worker 2 is ready.
Worker 3 spends 5 seconds on job 2 ...
Worker 1 spends 4 seconds on job 1 ...
Worker 2 spends 5 seconds on job 3 ...
Worker 1 spends 2 seconds on job 4 ...
Worker 3 spends 6 seconds on job 6 ...
Worker 2 spends 4 seconds on job 5 ...
Worker 1 spends 3 seconds on job 7 ...
Worker 2 spends 6 seconds on job 8 ...
Worker 1 spends 6 seconds on job 9 ...
Worker 3 spends 4 seconds on job 10 ...
Jobs done : [1, 3, 2, 1, 2, 3, 1, 2, 1, 3]
```

the setup: generating a queue of jobs

```
using Base.Threads

nbr = 10
jobs = rand((2, 3, 4, 5, 6), nbr)
println("The jobs : ", jobs)

nt = nthreads()
println("the number of threads : ", nt)

@threads for i=1:nt
    println("Worker ", threadid(), " is ready.")
end
```

the job index is an Atomic variable

```
jobidx = Atomic{Int}(1)
@threads for i=1:nt
    println("Worker ", threadid(), " is ready.")
    while true
        myjob = atomic_add!(jobidx, 1)
        if myjob > nbr
            break
        end
        println("Thread ", threadid(),
                " spends ", jobs[myjob], " seconds",
                " on job ", myjob, " ...")
        sleep(jobs[myjob])
        jobs[myjob] = threadid()
    end
end
println("Jobs done : ", jobs)
```

using an Atomic variable

The job index is accessed in a *thread safe* manner using an Atomic variable.

- The job index is declared and initialized to one:

```
jobidx = Atomic{Int}(1)
```

- Incrementing the job index:

```
myjob = atomic_add!(jobidx, 1)
```

  returns the current value of `jobidx` and increments the value of `jobidx` by one.

The thread safe manner means that accessing the value of the job index can done by only one thread at the same time.

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

processing a queue of jobs

The jobs we consider can be executed independently.

In the simulation, the queue is an array of integers.

Every job in the queue

1. has a unique job number; and
2. a computational cost, defined by an integer.

Job j has cost w_j. We execute `sleep`(w_j), to wait w_j seconds.

In the sequential processing of `N` jobs, we do

```
for(int j=0; j<N; j++) sleep(w[j]);
```

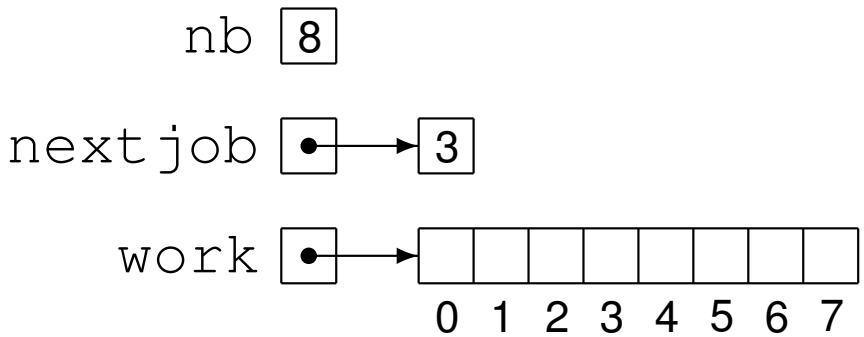
Typically, `N` is much larger than the number of threads.

representing a job queue

The state of the job queue is defined by

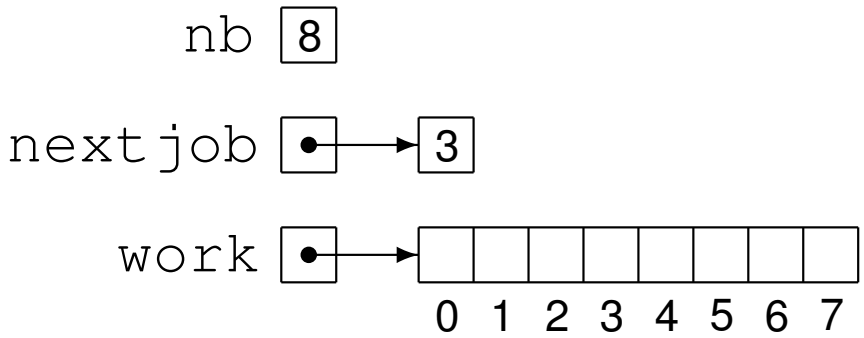
1. the number of jobs,
2. the index to the next job to be executed,
3. the work to be done by every job.

Variables in a program can be values or references to values.



values and references to values

Variables in a program can be values or references to values.



In C, the above picture is realized by the statements:

```
int nb = 8;
int *nextjob;
int *work;

*nextjob = 3;
work = (int*)calloc(nb, sizeof(int));
```

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

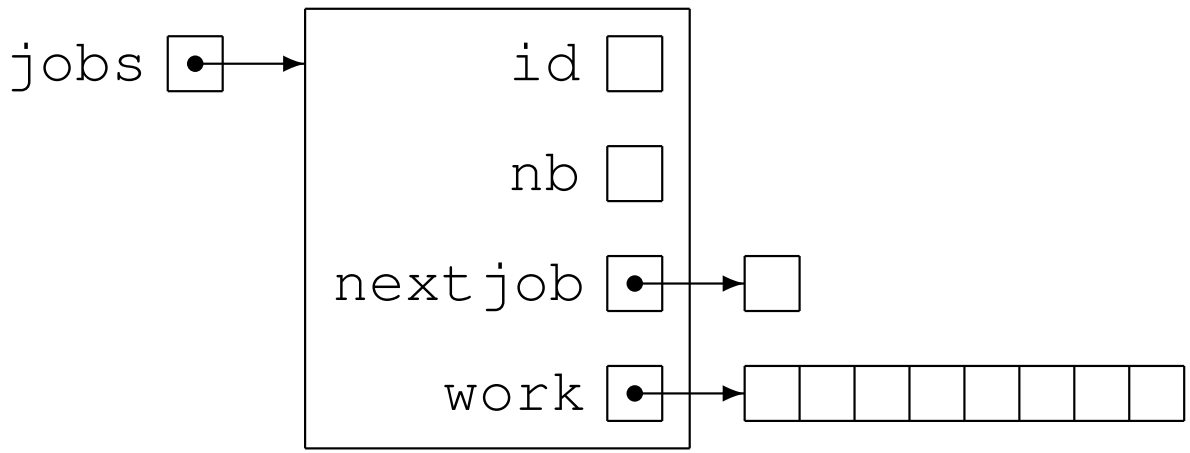
encapsulating references to shared data

Every thread has as values

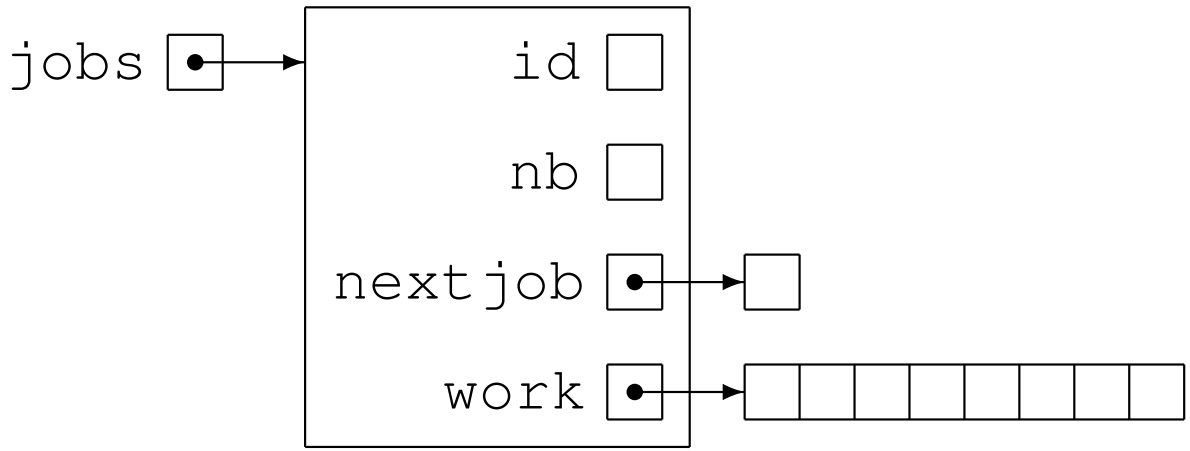
1. its thread identification number `id`; and
2. the number of jobs `nb`.

The shared data are

1. the reference to the next job; and
2. the cost for every job.

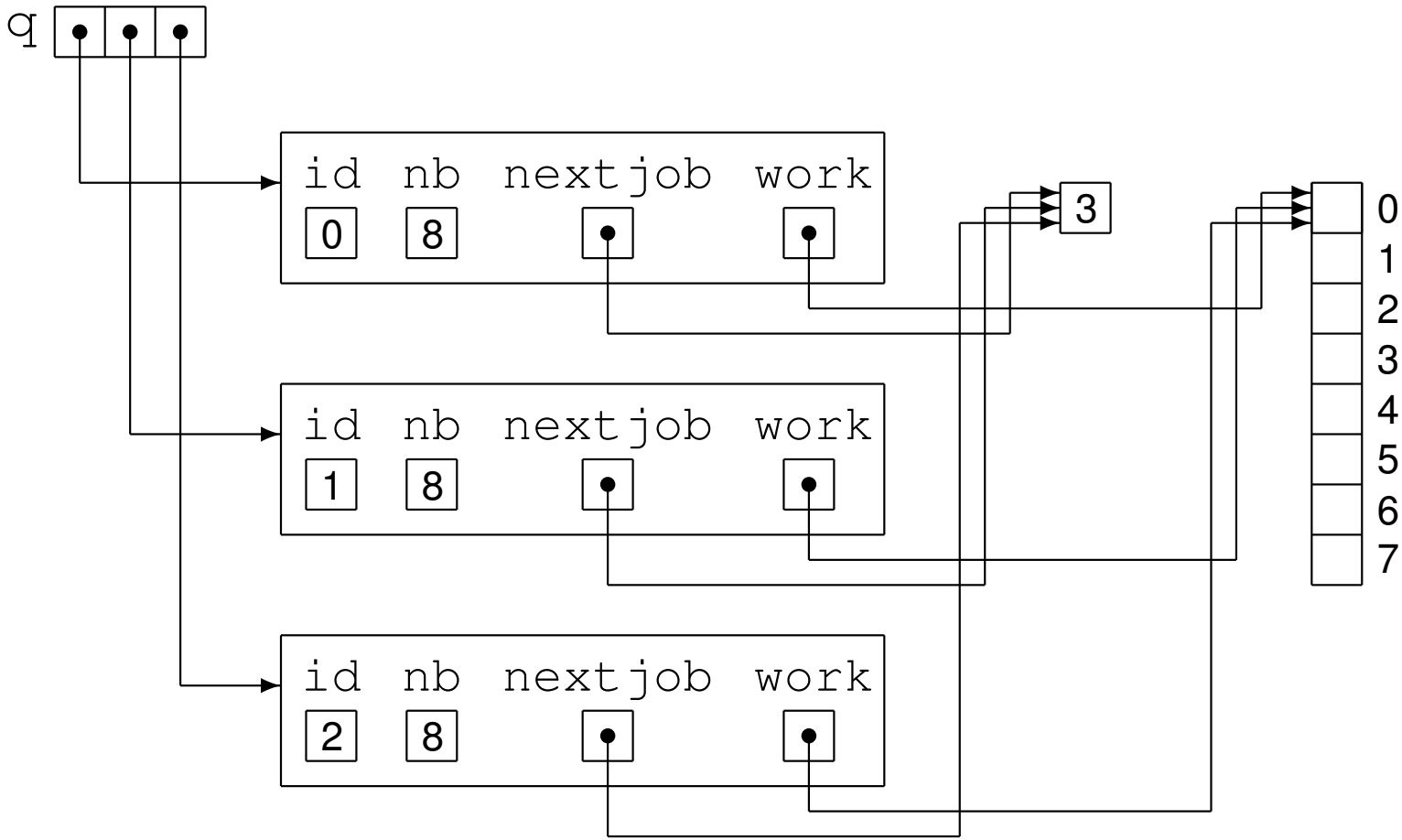


definition of the job queue

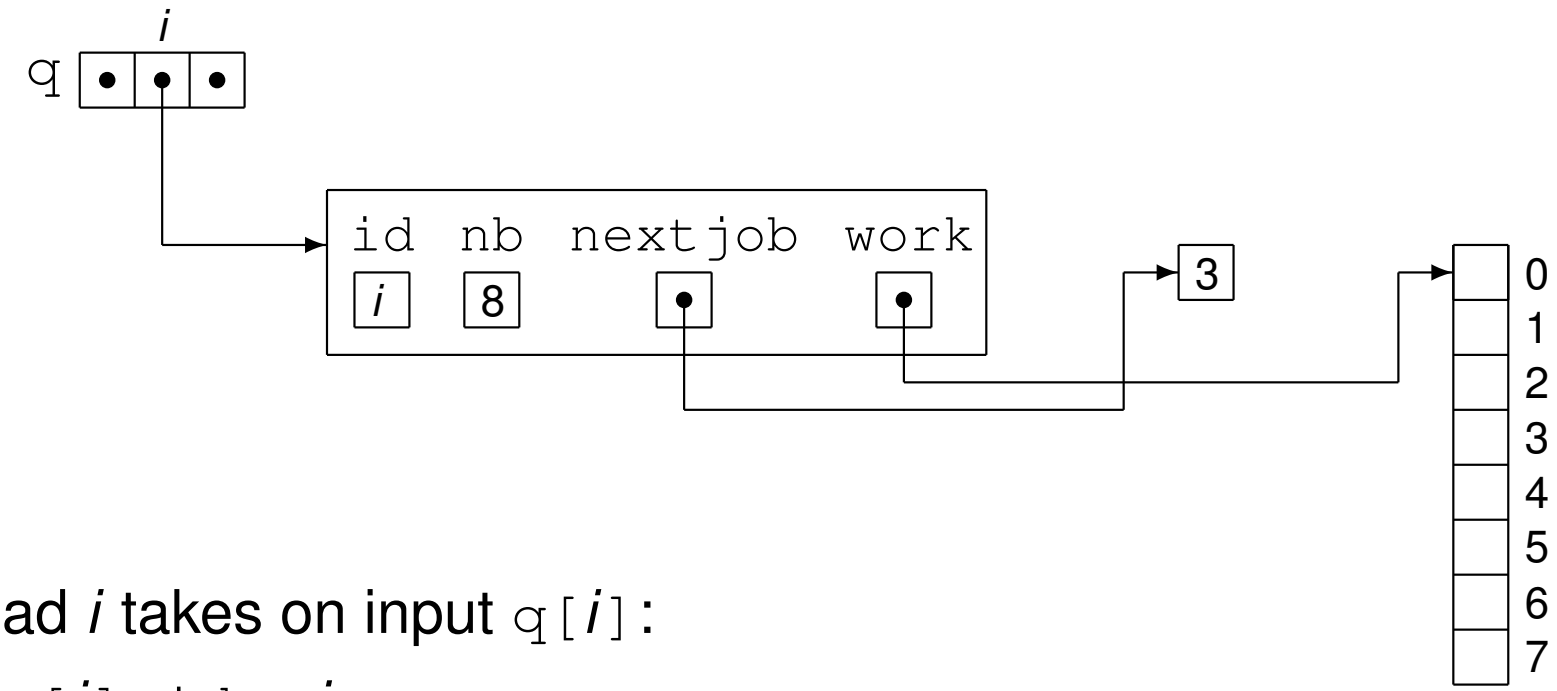



```
typedef struct
{
   int id;          /* identification label */
   int nb;          /* number of jobs */
   int *nextjob;    /* index of next job */
   int *work;       /* array of nb jobs */
} jobqueue;
```

sharing data between three threads



accessing shared data



Thread *i* takes on input `q[i]`:

1. `q[i].id` = *i*,
2. `q[i].nb` = 8,
3. `*q[i].nextjob` = 3,
4. `q[i].work[3]` defines the next job.

processing the jobs

```
void do_job ( jobqueue *q )
{
   int jobtodo;
   do
   {
      jobtodo = -1;
      int *j = q->nextjob;
      if(*j < q->nb) jobtodo = (*j)++;
      if(jobtodo == -1) break;
```

The `q->nextjob` is equivalent to `(*q).nextjob`.
The `jobtodo = (*j)++` dereferences `j`, assigns, and increments.

```
      int w = q->work[jobtodo];
      sleep(w);
   }
   while (jobtodo != -1);
}
```

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

processing the jobs with OpenMP

```
void do_job ( jobqueue *q )
{
   int jobtodo;
   do
   {
      jobtodo = -1;
      int *j = q->nextjob;

      #pragma omp critical
      if(*j < q->nb) jobtodo = (*j)++;

      if(jobtodo == -1) break;
      int w = q->work[jobtodo];
      sleep(w);
   }
   while (jobtodo != -1);
}
```

defining the parallel region

```
int process_jobqueue ( jobqueue *jobs, int nbt )
{
   jobqueue q[nbt];
   int i;

   for(i=0; i<nbt; i++)
   {
      q[i].nb = jobs->nb;
      q[i].id = i;
      q[i].nextjob = jobs->nextjob;
      q[i].work = jobs->work;
   }
   #pragma omp parallel
   {
      i = omp_get_thread_num();
      do_job(&q[i]);
   }
   return *(jobs->nextjob);
}
```

the main program

```
int main ( int argc, char* argv[] )
{
   int njobs,done,nbthreads;
   jobqueue *jobs;

   /* prompt for njobs and nbthreads */

   jobs = make_jobqueue(njobs);

   omp_set_num_threads(nbthreads);

   done = process_jobqueue(jobs,nbthreads);

   printf("done %d jobs\n", jobs->nb);

   return 0;
}
```

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

POSIX threads

For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.

POSIX = Portable Operating System Interface

Implementations of this POSIX threads programming interface are referred to as POSIX threads, or Pthreads.

```
$ gcc -v
... output omitted ...
Thread model: posix
... output omitted ...
```

In a C program we just insert

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

and compilation may require the switch `-pthread`

```
$ gcc -pthread program.c
```

the function each thread executes

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

void *say_hi ( void *args );
/*
 * Every thread executes say_hi.
 * The argument contains the thread id. */

int main ( int argc, char* argv[] ) { ... }

void *say_hi ( void *args )
{
   int *i = (int*) args;
   printf("hello world from thread %d!\n",*i);
   return NULL;
}
```

running `hello_pthreads`

```
$ make hello_pthreads
gcc -o hello_pthreads hello_pthreads.c

$ ./hello_pthreads
How many threads ? 5
creating 5 threads ...
waiting for threads to return ...
hello world from thread 0!
hello world from thread 2!
hello world from thread 3!
hello world from thread 1!
hello world from thread 4!
$
```

the main program

```
int main ( int argc, char* argv[] ) {
   printf("How many threads ? ");
   int n; scanf("%d",&n);
   {
      pthread_t t[n];
      pthread_attr_t a;
      int i,id[n];
      printf("creating %d threads ...\n",n);
      for(i=0; i<n; i++)
      {
         id[i] = i;
         pthread_attr_init(&a);
         pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
      }
      printf("waiting for threads to return ...\n");
      for(i=0; i<n; i++) pthread_join(t[i],NULL);
   }
   return 0;
}
```

avoiding sharing of data between threads

To each thread we pass its unique identification label.
To `say_hi` we pass the address of the label.

With the array `id[n]` we have `n` distinct addresses:

```
pthread_t t[n];
pthread_attr_t a;
int i,id[n];
for(i=0; i<n; i++)
{
   id[i] = i;
   pthread_attr_init(&a);
   pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
}
```

Passing `&i` instead of `&id[i]` gives to every thread the same address, and thus the same identification label.

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

using Pthreads in 3 steps

1. Declare threads of type `pthread_t`
   and attribute(s) of type `pthread_attri_t`.

2. Initialize the attribute `a` as `pthread_attr_init(&a);`
   and create the threads with `pthreads_create` providing
   1. the address of each thread,
   2. the address of an attribute,
   3. the function each thread executes, and
   4. an address with arguments for the function.

   Variables are shared between threads if the same address is passed as argument to the function the thread executes.

3. The creating thread waits for all threads to finish
   using `pthread_join`.

The Crew of Threads Model

1. The Work Crew Model
   - programming shared memory parallel computers
   - multiple threads cooperating on a queue of jobs
   - a crew of workers with Julia
2. Processing a Queue of Jobs
   - the sequential setup
   - sharing data between threads
   - processing the jobs with OpenMP
3. the POSIX Threads Programming Interface
   - our first program with Pthreads
   - attributes, Pthread creating and joining
   - implementing a critical section with `mutex`

using `mutex`

Three steps to use a `mutex` (mutual exclusion):

1. initialization

   ```
   pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;
   ```

2. request a lock

   ```
   pthread_mutex_lock(&L);
   ```

3. release the lock

   ```
   pthread_mutex_unlock(&L);
   ```

processing jobs with pthreads

```
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;

int main ( int argc, char* argv[] )
{
   printf("How many jobs ? ");
   int njobs; scanf("%d",&njobs);
   jobqueue *jobs = make_jobqueue(njobs);
   if(v > 0) write_jobqueue(jobs);

   printf("How many threads ? ");
   int nthreads; scanf("%d",&nthreads);
   int done = process_jobqueue(jobs,nthreads);
   printf("done %d jobs\n",done);
   if(v>0) write_jobqueue(jobs);

   return 0;
}
```

the function do_job

```
void *do_job ( void *args )
{
   jobqueue *q = (jobqueue*) args;
   int dojob;
   do
   {
      dojob = -1;
      /* code omitted */
   } while (dojob != -1);

   if(v>0) printf("thread %d is finished\n",q->id);

   return NULL;
}
```

the `do while` loop

```
do
{
   dojob = -1;
   if(v > 0) printf("thread %d requests lock ...\n",q->id);
   pthread_mutex_lock(&read_lock);
   int *j = q->nextjob;
   if(*j < q->nb) dojob = (*j)++;
   if(v>0) printf("thread %d releases lock\n",q->id);
   pthread_mutex_unlock(&read_lock);
   if(dojob == -1) break;
   if(v>0) printf("*** thread %d does job %d ***\n",
                  q->id,dojob);
   int w = q->work[dojob];
   if(v>0) printf("thread %d sleeps %d seconds\n",q->id,w);
   q->work[dojob] = q->id; /* mark job with thread label */
   sleep(w);
} while (dojob != -1);
```

Bibliography

Online documents on Pthreads:

- Guide to the POSIX Threads Library, April 2001,
  by Compaq Computer Corporation, Houston Texas.
- Threading Programming Guide,
  Mac OS X Developer Library, 2010.

Summary + Exercises

We covered dynamic load assignment for multitreaded programs.

Exercises:

1. Modify the `hello world!` program with Pthreads so that the master thread prompts the user for a name which is used in the greeting displayed by thread 5. Note that only one thread, the one with number 5, greets the user.
2. Consider the Monte Carlo simulations we have developed with MPI for the estimation of π. Write a version with Julia, or OpenMP, or Pthreads and examine the speedup.

two extra exercises

3. Consider the computation of the Mandelbrot set as implemented in the program `mandelbrot.c` of lecture 7. Write code (with Julia, or OpenMP, or Pthreads) for a work crew model of threads to compute the grayscales. Does the grain size matter? Compare the running time of your program with your MPI implementation.
4. For some number `N`, array `x`, function `f`, consider:

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
 {
    for(i=0; i<N; i++) x[i] = f(i);
 }
```

   Define the simulation of the dynamic load balancing with the job queue using `schedule(dynamic)`.