Multithreaded Programming with Posix Threads

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl

University of Amsterdam

Concurrency and Parallel Programming
Slides by: Dr. Clemens Grelck.

Multithreaded Programming with Posix Threads

Targeted Computing Systems

Small scale:

- dual-core processors
- quad-core processors
- your laptop or desktop

Targeted Computing Systems

Small scale:

- dual-core processors
- quad-core processors
- your laptop or desktop

Medium scale:

- bread-and-butter servers
- 2–4 processors
- 4–8 cores each

Targeted Computing Systems

Small scale:

- dual-core processors
- quad-core processors
- your laptop or desktop

Medium scale:

- bread-and-butter servers
- 2–4 processors
- 4–8 cores each

Large scale:

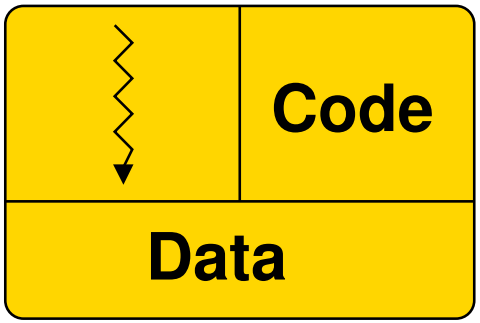
- high-end compute servers
- 48/64-core x86 systems
- 64-core 8-fold hw-multithreaded Oracle Niagara

Targeted Computing Systems

Common characteristic: **hardware shared address space**

From Sequential to Multithreaded Programming

Classical process:

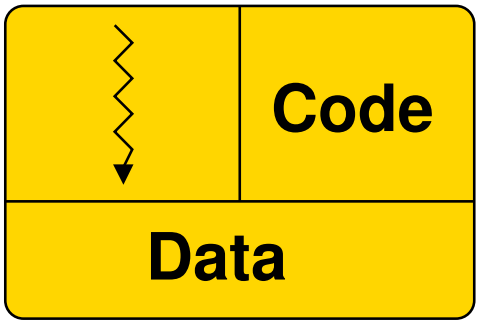


Characteristics:

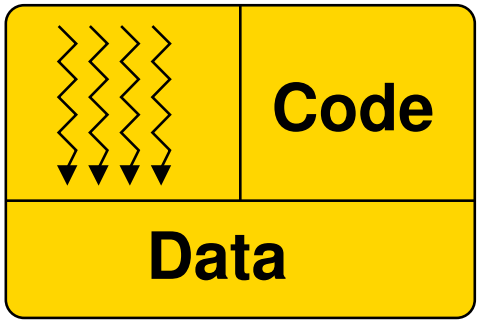
- One code
- One program counter
- One register set
- One runtime stack
- One data heap

From Sequential to Multithreaded Programming

Classical process:



Multithreaded Process:



Characteristics:

- One code
- One program counter
- One register set
- One runtime stack
- One data heap

Characteristics:

- One code
- Multiple program counters
- Multiple register sets
- Multiple runtime stacks
- One data heap

Multithreading APIs

Historically:

- Different APIs for different operating systems
- Vendor- and independent designs and implementations

Our focus: ANSI / IEEE POSIX standard:

- First standard in 1995
- Supported by all serious operating systems
- Defined for C programming language

Other threading models:

- Microsoft threads ...
- Native language support: Java, C#, ...

Multithreaded Programming with Posix Threads

Thread Creation

Characteristics:

- Only one thread at program startup.
- All threads are created dynamically.
- Threads may terminate prior to process termination.
- Process terminates when initial thread terminates.
- All threads terminate when process terminates.

Function pthread_create:

```
int pthread_create(
  pthread_t *thread_id,              // OUT : thread identifier
  pthread_attr_t *attributes,        // IN  : thread attributes
  void *(*start_routine)(void*),     // IN  : code to execute
  void *argument)                    // IN  : for start routine
```

Thread Creation Example: Multithreaded Hello World

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

void *HelloWorld( void *a)
{
  printf( "Hello World\n");
  return NULL;
}

int main()
{
  pthread_t thread_ids[MAX];
  int i;

  for (i=0; i<MAX; i++) {
    pthread_create( &thread_ids[i],    // returned thread ids
                    NULL,              // default attributes
                    &HelloWorld,       // start routine
                    NULL);             // thread argument
  }
  return 0;
}
```

Compiling and Linking Multithreaded Programs

Compiling:

- `cc -D_POSIX_C_SOURCE=199506L ....`
- `cc -D_POSIX_C_SOURCE=200112L ....`
- Compile each source file with one of the above flags !
- June 1995: initial standard
- December 2001: various extensions

Linking:

- `cc .... -lpthread`
- Link with libpthread.so

Compiling and Linking Multithreaded Programs

Compiling:

- `cc -D_POSIX_C_SOURCE=199506L ....`
- `cc -D_POSIX_C_SOURCE=200112L ....`
- Compile each source file with one of the above flags !
- June 1995: initial standard
- December 2001: various extensions

Linking:

- `cc ....  -lpthread`
- Link with libpthread.so

Or when using gcc:

- `gcc ....  -pthread`

Thread Creation Example: Multithreaded Hello World

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

void *HelloWorld( void *a)
{
  printf( "Hello World\n");
  return NULL;
}

int main()
{
  pthread_t thread_ids[MAX];
  int i;

  for (i=0; i<MAX; i++) {
    pthread_create( &thread_ids[i],    // returned thread ids
                    NULL,              // default attributes
                    &HelloWorld,       // start routine
                    NULL);             // thread argument
  }
  return 0;
}
```

How many lines of text does the program produce ?

Waiting for Threads

Characteristics:

- Any thread may wait for termination of any other thread.
- This is called "joining" a thread.
- Terminated threads remain zombies until joined by someone.
- Any thread may be joined at most once.

Function pthread_join:

```
int pthread_join(
  pthread_t thread_id,    // IN: thread to join
  void **return_value)    // OUT: return value of start routine
```

Thread Creation Example

Multithreaded Hello World Reloaded:

```
int main()
{
  pthread_t thread_ids[MAX];
  int i;
  void *results[MAX];

  for (i=0; i<MAX; i++) {
    pthread_create( &thread_ids[i],    // thread ids
                    NULL,              // default attributes
                    &HelloWorld,       // start routine
                    NULL);             // thread argument
  }

  for (i=0; i<MAX; i++) {
    pthread_join( thread_ids[i],       // thread to wait for
                  &results[i]);        // result of thread
  }

  return 0;
}
```

Thread Argument Passing

Let HelloWorld print thread number:

```
void *HelloWorld( void *a)
{
  int* tid_p = (int*) a;  // cast arg to int pointer
  int  tid   = *tid_p;    // dereference int pointer

  printf( "This is thread %d\n", tid);

  return a;
}
```

Thread Argument Passing

Let HelloWorld print thread number:

```
int main()
{
  pthread_t thread_ids[MAX];
  int i, *num;
  void *result;

  for (i=0; i<MAX; i++) {
    num = (int*) malloc( sizeof( int));
    *num = i;
    pthread_create( &thread_ids[i], NULL,
                    &HelloWorld, num);
  }

  for (i=0; i<MAX; i++) {
    pthread_join( thread_ids[i], &result);
    free( result);
  }

  return 0;
}
```

Thread Attributes

Creation and finalisation:

```
int
pthread_attr_init(          // initialize opaque data
  pthread_attr_t *attr)     // structure to default values

int
pthread_attr_destroy(
  pthread_attr_t *attr)     // remove opaque data structure
```

Usage pattern:

```
pthread_attr_t attributes;                  // declare attr struct
...
pthread_attr_init( &attributes);            // init attr struct
...
pthread_create( ..., &attributes, ...);     // use attr struct
...
pthread_attr_destroy( &attributes);         // remove attr struct
```

Thread Attributes

Example: detach state

- PTHREAD_CREATE_JOINABLE
  can be joined (default)
- PTHREAD_CREATE_DETACHED
  cannot be joined, fire and forget

Functions for manipulation of attributes:

```
int pthread_attr_setdetachstate(
  pthread_attr_t *attributes,      /* opaque attrib struct */
  int detachstate);                /* desired detach state */

int pthread_attr_getdetachstate(
  pthread_attr_t *attributes,      /* opaque attrib struct */
  int *detachstate);               /* OUT: detach state    */
```

Thread Attributes

Example: thread scope

- PTHREAD_SCOPE_PROCESS
  cooperative threads inside process (default)
- PTHREAD_SCOPE_SYSTEM
  pre-emptive threads scheduled by operating system

Functions for manipulation of attributes:

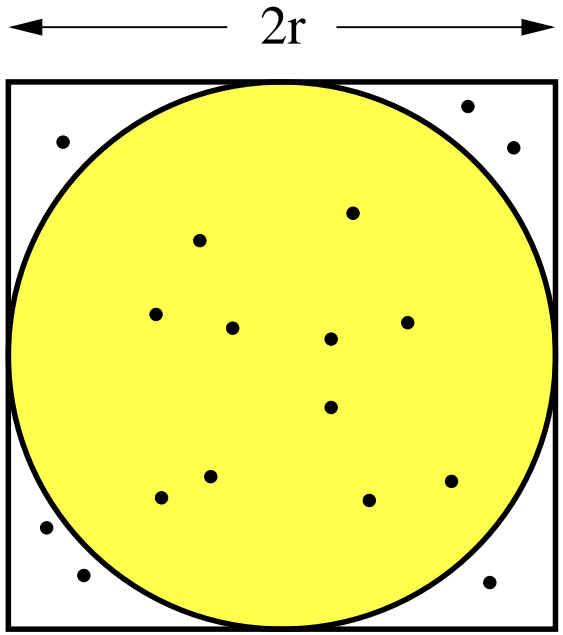
```
int pthread_attr_setscope(
  pthread_attr_t *attributes,     /* opaque attrib struct */
  int scope);                     /* desired scope value  */

int pthread_attr_getscope(
  pthread_attr_t *attributes,     /* opaque attrib struct */
  int *scope);                    /* OUT: scope state     */
```

Multithreaded Programming with Posix Threads

Example: Approximation of PI

Outline of Algorithm:

1. Inscribe circle of radius r in square of edge length $2r$.
2. Randomly generate A_s points in the square.
3. Determine number of points A_c which are within circle.
4. Compute Pi:
$$pi := 4 \times \frac{A_c}{A_s}$$
5. More points, better approximation.
6. Known as Monte Carlo simulation.



Monte Carlo Approximation of PI

```
double approx_pi( int sample_points)
{
  int hits;  double pi;

  total_hits = simulate( sample_points);
  pi = 4.0 * ((double) hits) / ((double) sample_points);
  return pi;
}

int simulate( int sample_points)
{
  int seed=1, hits=0, i;
  double rand_no_x, rand_no_y;

  for (i=0; i<sample_points; i++) {
    rand_no_x = (double) rand_r(&seed) / (double) RAND_MAX;
    rand_no_y = (double) rand_r(&seed) / (double) RAND_MAX;

    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
         (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25) {
      hits++;
  } }
  return hits;
}
```

Multithreaded Monte Carlo Approximation of PI

```
double approx_pi( int num_threads, int sample_points)
{
  int i, total_hits, hits[MAX_THREADS];
  pthread_t p_threads[MAX_THREADS];
  pthread_attr_t attr; double pi;

  pthread_attr_init( &attr);
  pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM);

  sample_points_per_thread = sample_points / num_threads;

  for (i=0; i<num_threads; i++) {
    hits[i] = i;
    pthread_create( &p_threads[i], &attr, simulate, &hits[i]); }

  total_hits = 0;
  for (i=0; i<num_threads; i++) {
    pthread_join( p_threads[i], NULL);
    total_hits += hits[i]; }

  pi = 4.0 * ((double) total_hits) / ((double) sample_points);

  return pi;
}
```

Multithreaded Monte Carlo Approximation of PI

```
void *simulate( void *s)
{
  int seed, i, *hit_pointer, local_hits;
  double rand_no_x, rand_no_y;

  hit_pointer = (int *)s;
  seed = *hit_pointer;
  local_hits = 0;

  for (i=0; i<sample_points_per_thread; i++) {
    rand_no_x = (double) rand_r(&seed) / (double) RAND_MAX;
    rand_no_y = (double) rand_r(&seed) / (double) RAND_MAX;

    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
         (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25) {
      local_hits++;
    }
  }

  *hit_pointer = local_hits;

  return NULL;
}
```

Multithreaded Programming with Posix Threads

Private vs Shared Data

Private Data:

- Local variables of functions and blocks in general.
- Exception: Their address is provided as thread argument.
- Problem here: Stack frame of creator thread must survive created thread.

Private vs Shared Data

Private Data:

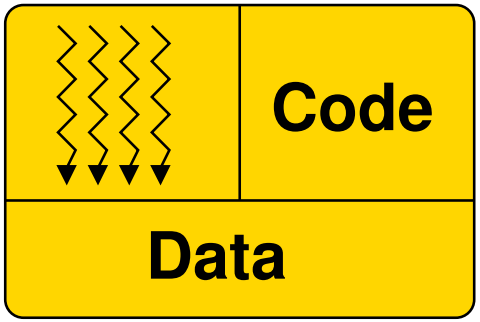
- Local variables of functions and blocks in general.
- Exception: Their address is provided as thread argument.
- Problem here: Stack frame of creator thread must survive created thread.

Shared Data:

- All global variables.
- Any allocated heap structure provided its address is made accessible to multiple threads (e.g. via global variable)

Towards the Mutual Exclusion Problem

Multithreaded Process:



Characteristics:

- One code
- Multiple program counters
- Multiple register sets
- Multiple runtime stacks
- **One data heap**

Crucial question:

What happens if multiple threads operate on the same data ?

Example: Task Queue

Runtime representation:

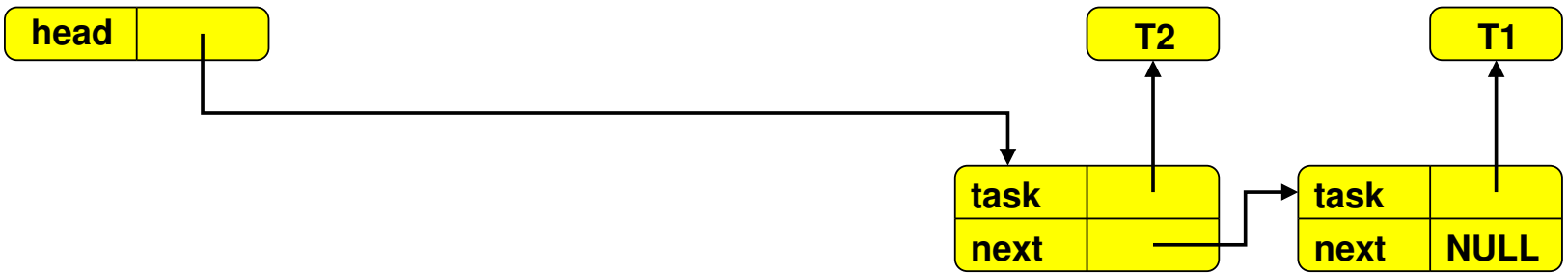
Programming interface:

```
extern void add_task( task_t *task);

extern task_t * get_task( void);
```

Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;
```

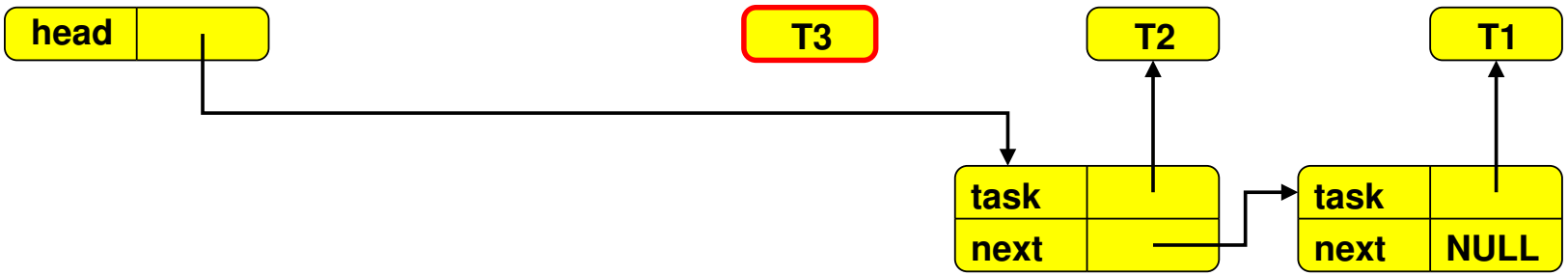
```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;

---------------------------
...
add_task( T3);
...
```

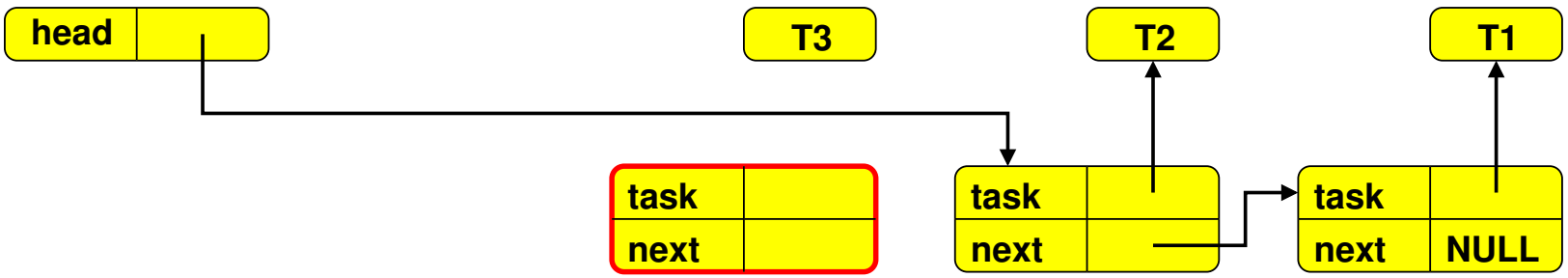
```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;

---------------------------
...
add_task( T3);
...
```

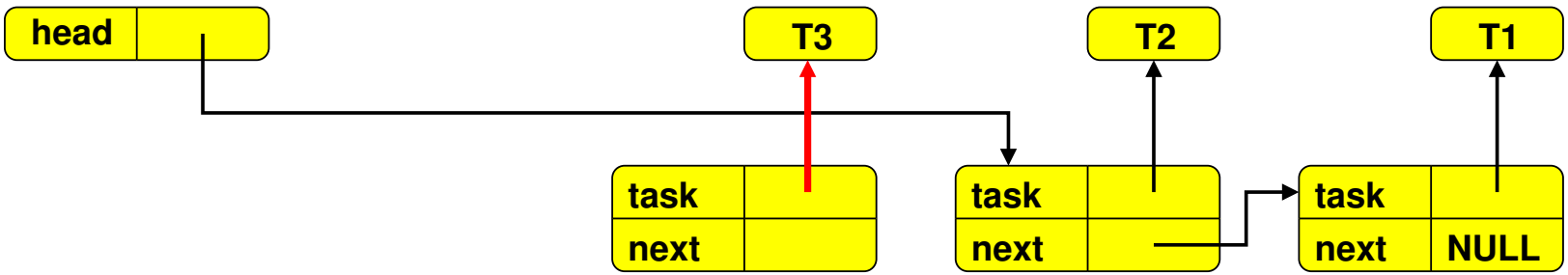
```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;

---------------------------
...
add_task( T3);
...
```

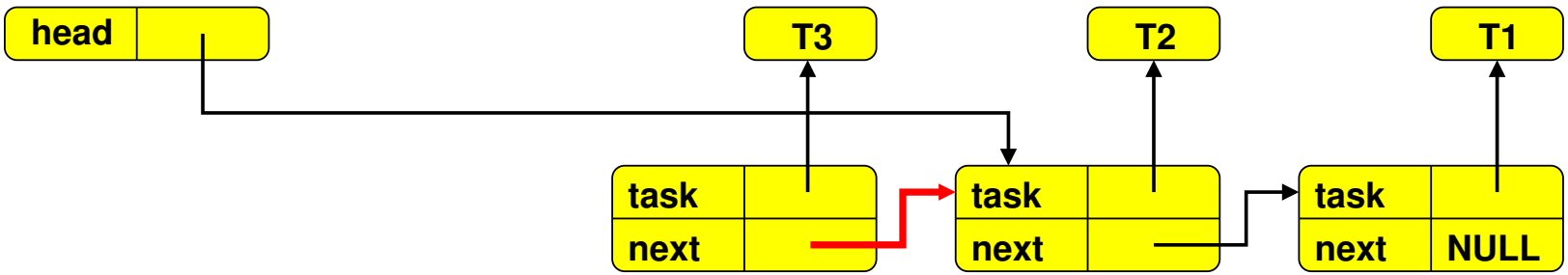
```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;

---------------------------
...
add_task( T3);
...
```

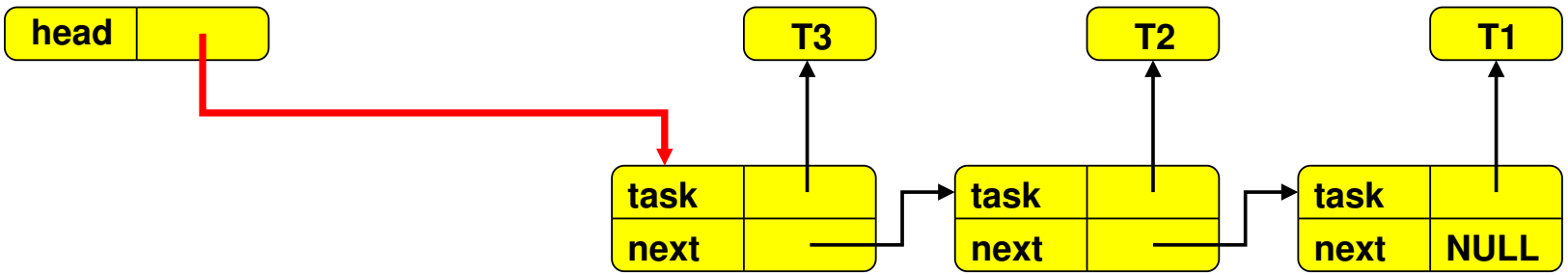
```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Adding one more Task

```
struct task_q_t {
  task_t *task;
  struct task_q_t *next;
}

static struct task_q_t
 *head = NULL;

---------------------------
...
add_task( T3);
...
```

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



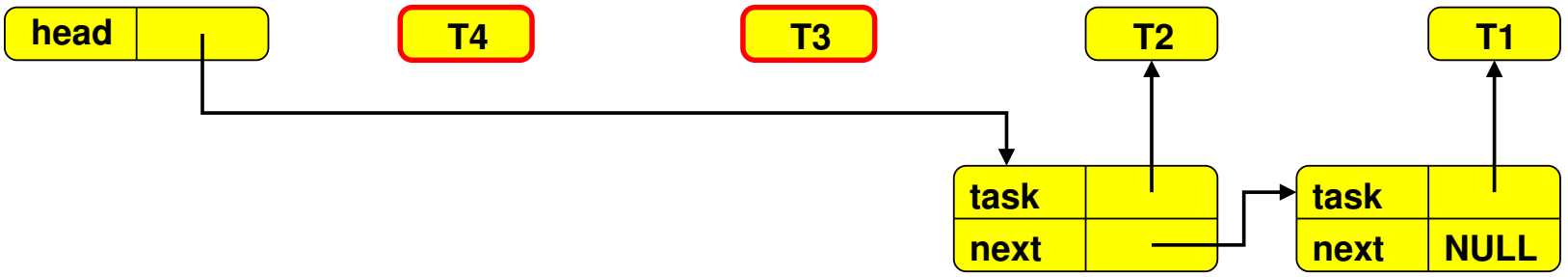
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



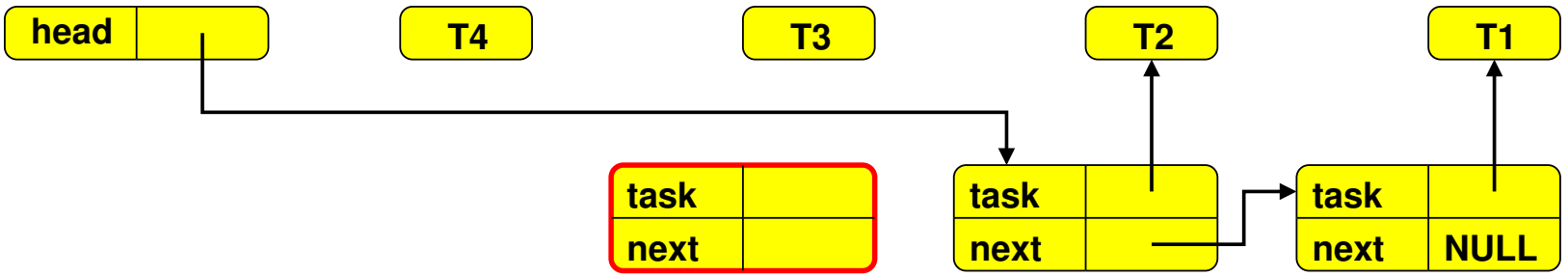
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



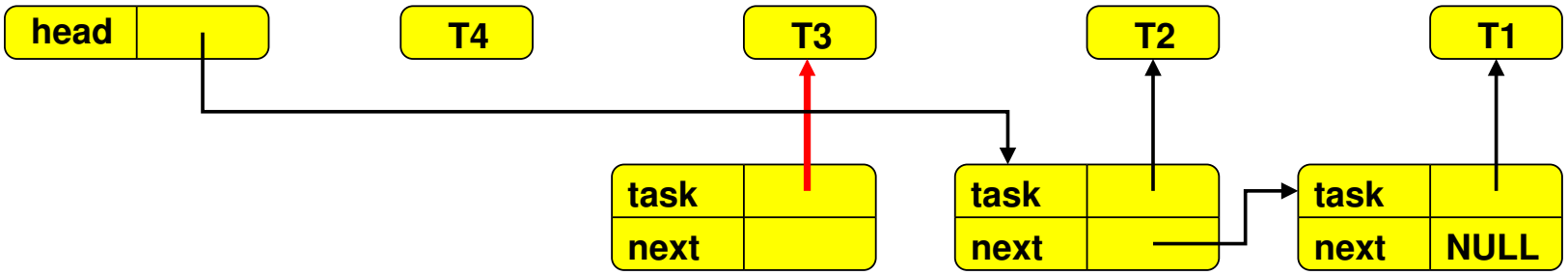
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



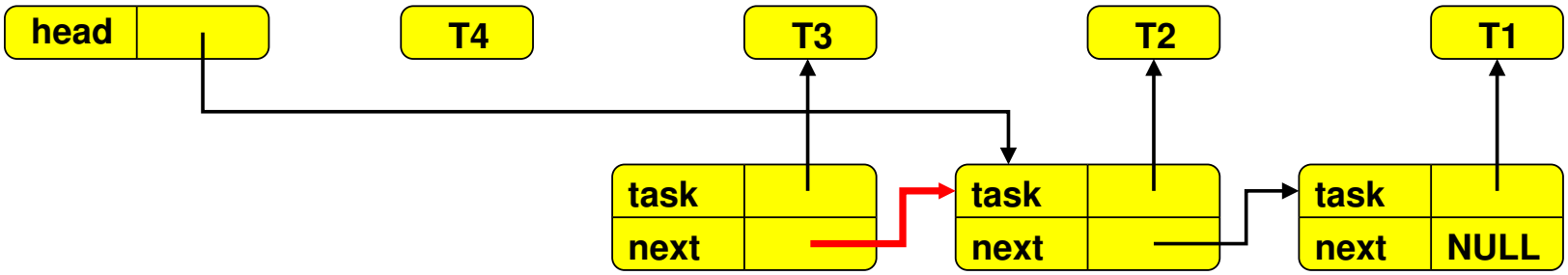
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



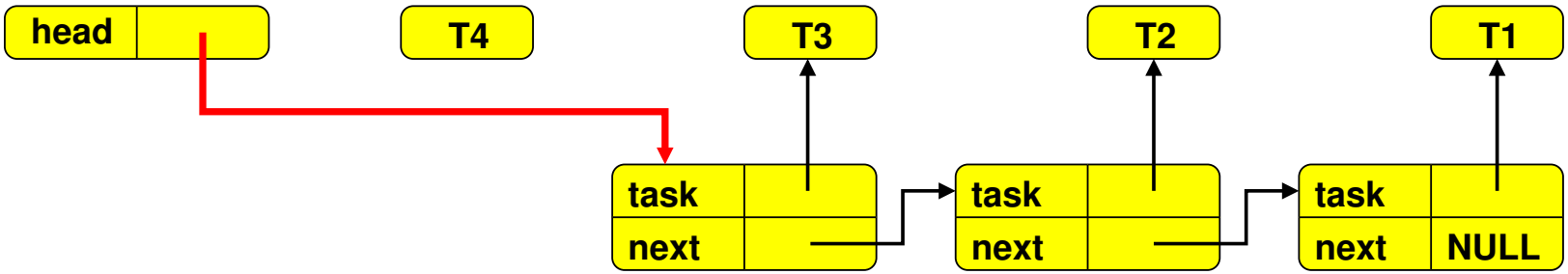
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



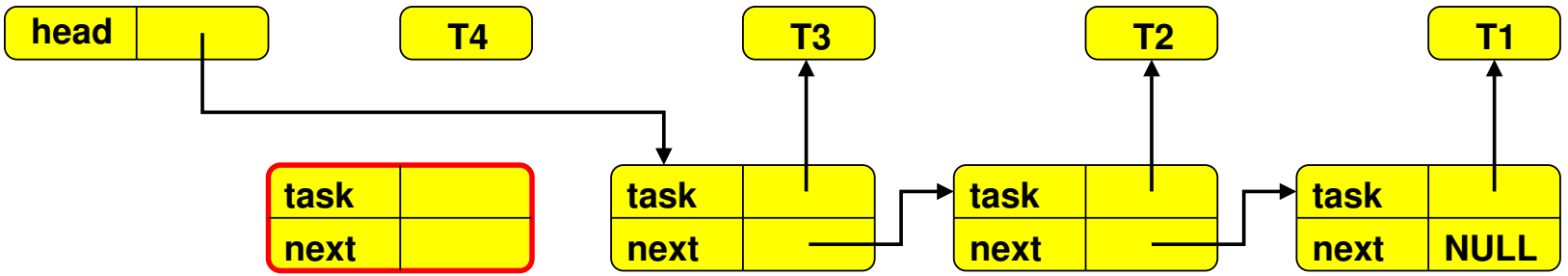
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



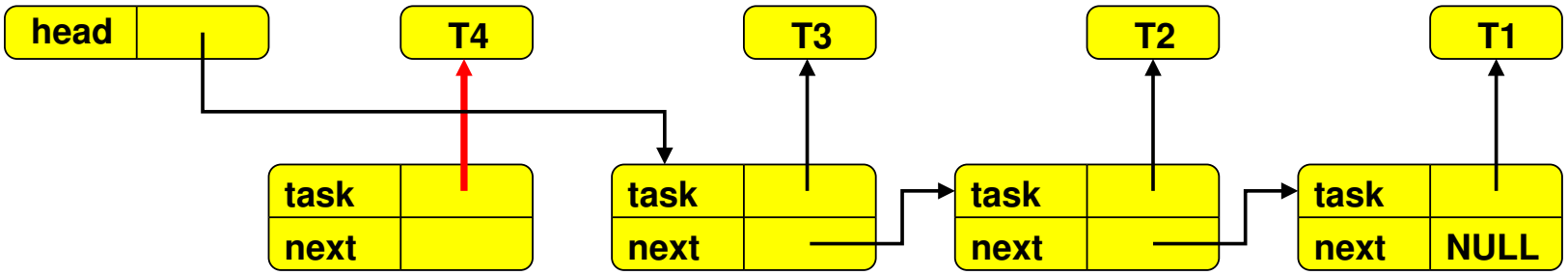
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



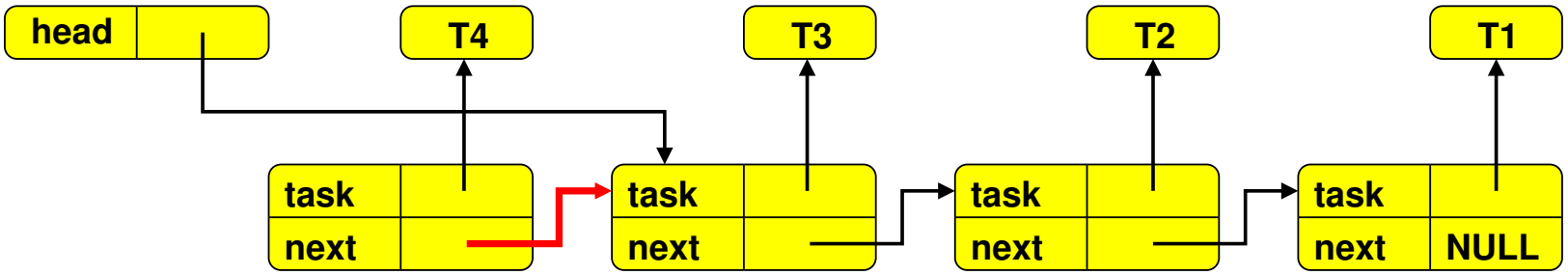
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



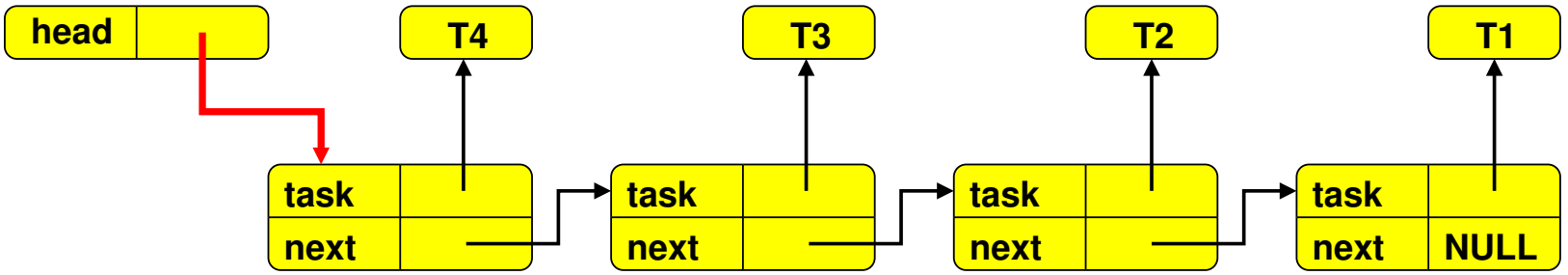
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



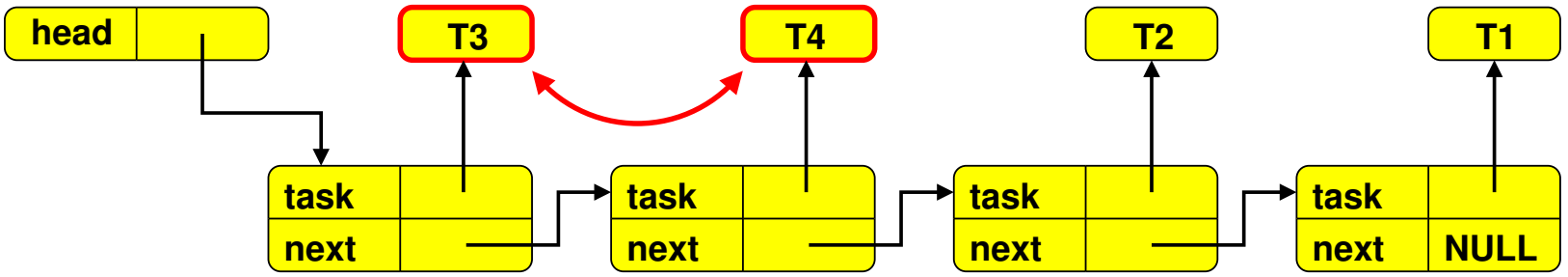
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



Non-deterministic program behaviour !!

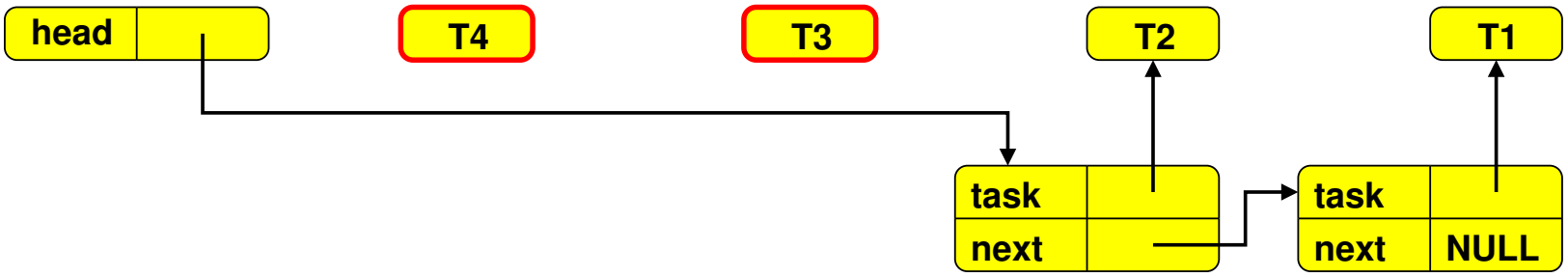
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



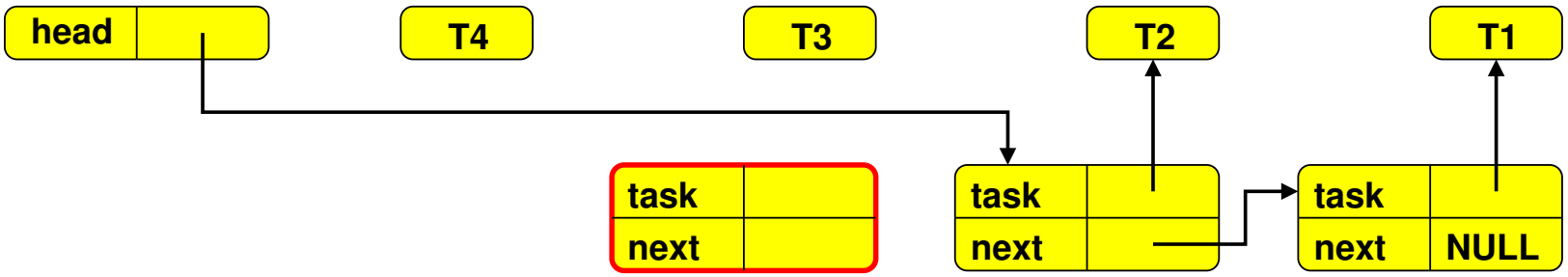
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



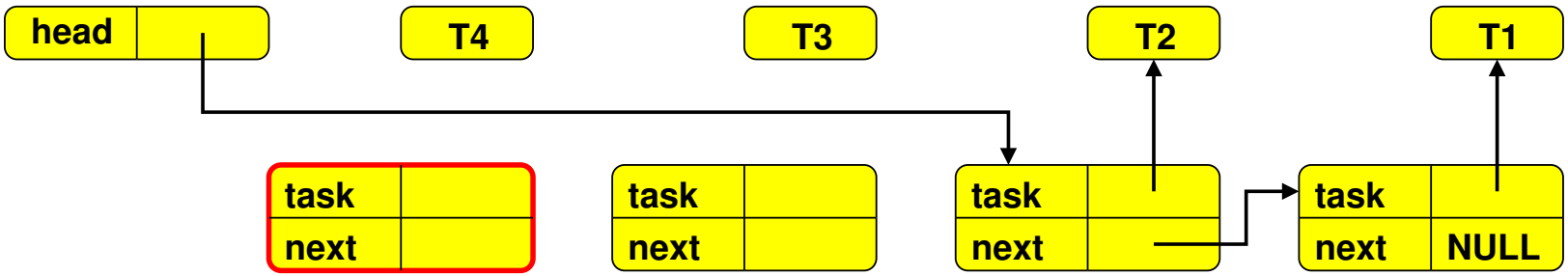
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



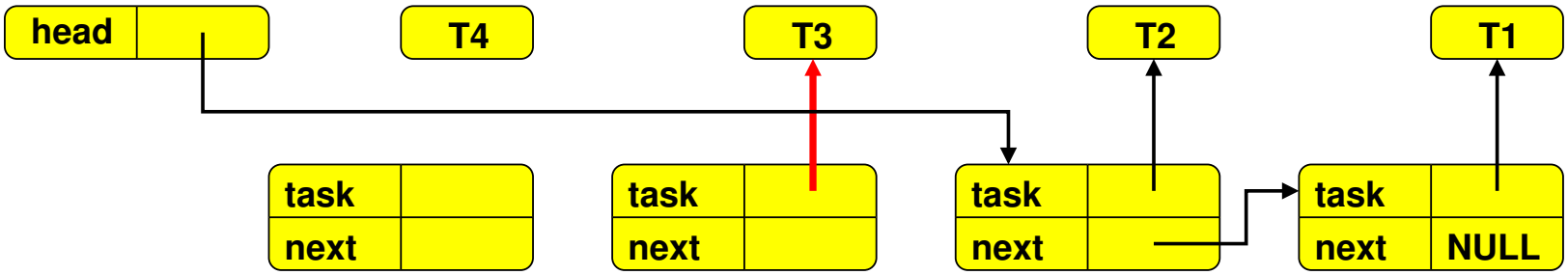
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



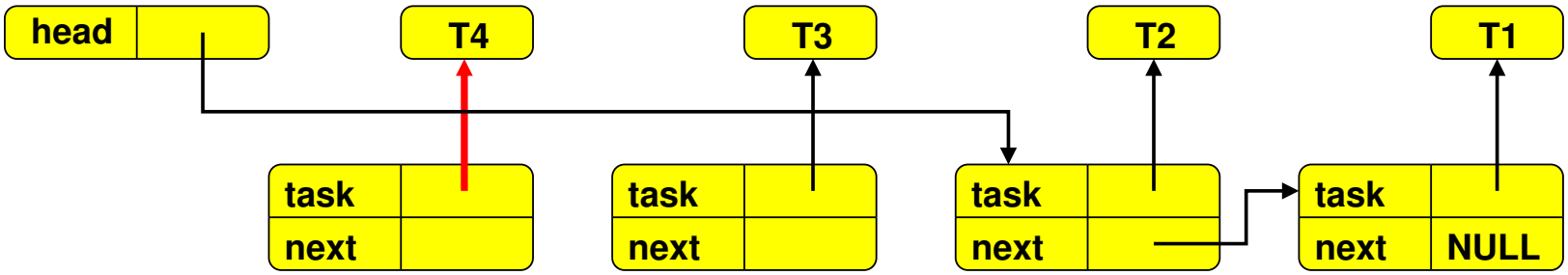
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



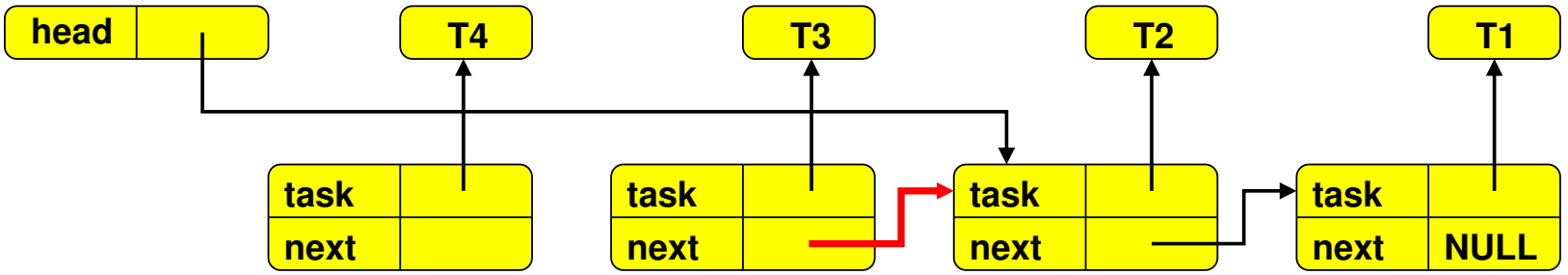
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



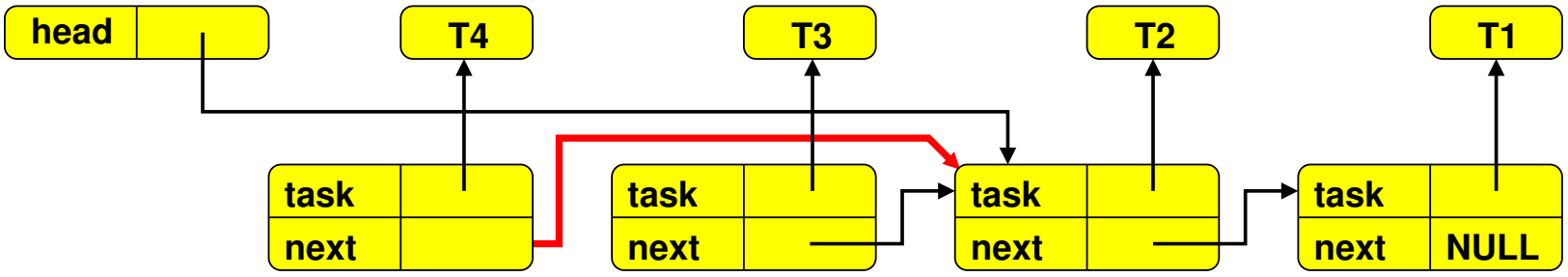
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



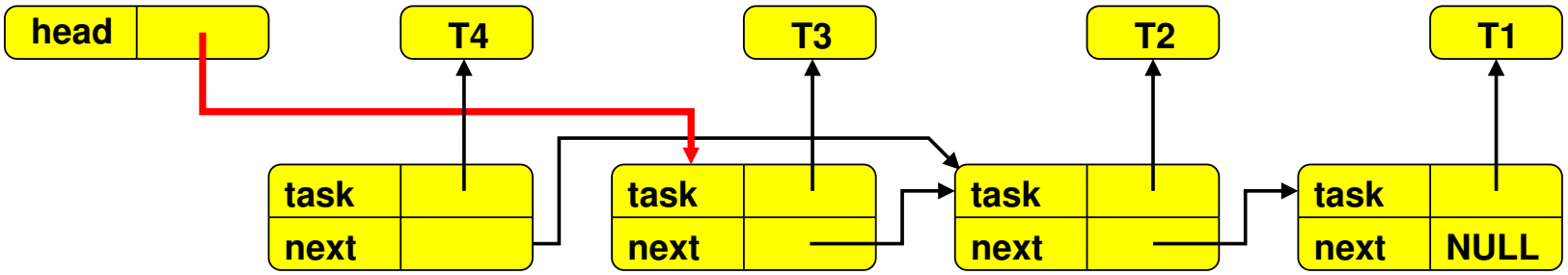
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



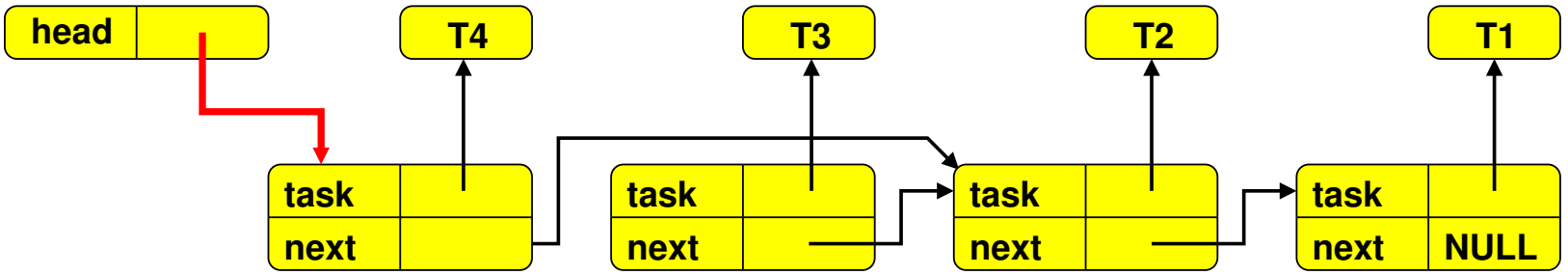
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



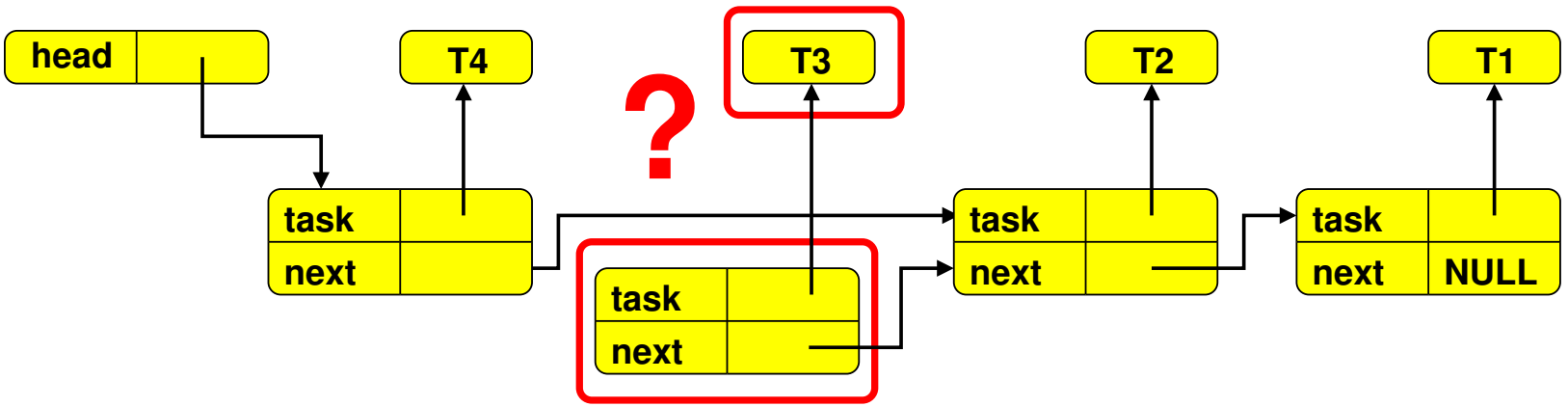
Adding one more Task: 2 Threads

Thread 1: `add_task(T3);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```

Thread 2: `add_task(T4);`

```
void add_task( task_t *task)
{
  task_q_t *newcell;

  newcell = malloc(...);
  newcell -> task = task;
  newcell -> next = head;
  head = newcell;
}
```



One task gets lost: non-deterministic and incorrect

Race Condition / Data Race

Definition:

- A *race condition / data race* exists if the behaviour (the meaning) of a program depends on the execution order of program parts (threads) whose temporal behaviour is beyond control.

Race Condition / Data Race

Definition:

- A *race condition / data race* exists if the behaviour (the meaning) of a program depends on the execution order of program parts (threads) whose temporal behaviour is beyond control.

Origin of term:

- Electronics
- Two electric signals race against each other.
- Arrival order of input signals at gate determines output signal.

Critical Regions

Definition:

- Sequence of statements which MUST be executed without interleaving.

Thread 1:

```
void insert( int newval)
{
  intlist *newcell;

  newcell = malloc(...);
  newcell->val = newval
  critical {
    newcell->next = head;
    head = newcell;
  }
}
```

Thread 2:

```
void insert( int newval)
{
  intlist *newcell;

  newcell = malloc(...);
  newcell->val = newval;
  critical {
    newcell->next = head;
    head = newcell;
  }
}
```

Implementing Critical Regions with Mutex Locks

Global:

```
typedef struct list { int val;
                      struct list *next } list;

list *head = NULL;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 1:

```
void insert( int newval)
{
  intlist *newcell;

  newcell = malloc(...);
  newcell->val = thread_id;
  pthread_mutex_lock( &lock);
    newcell->next = head;
    head = newcell;
  pthread_mutex_unlock( &lock);
}
```

Thread 2:

```
void insert( int newval)
{
  intlist *newcell;

  newcell = malloc(...);
  newcell->val = thread_id;
  pthread_mutex_lock( &lock);
    newcell->next = head;
    head = newcell;
  pthread_mutex_unlock( &lock);
}
```

Mutex Locks at a Glance

Properties:

- Mutex locks are abstract data objects.
- Only one thread at a time may hold a mutex lock.
- Threads block upon locking if lock is unavailable.
- Locking / unlocking are guaranteed to be atomic.
- No fairness on waiting threads upon unlocking.
- Only owner of lock can unlock.
- Re-locking by owner causes deadlock.

More Mutex Lock Operations

More functions:

- Dynamic initialisation with attributes:

```
int pthread_mutex_init( pthread_mutex_t      *lock,
                        pthread_mutexattr_t  *attributes);
```

- Dynamic de-allocation:

```
int pthread_mutex_destroy( pthread_mutex_t *lock);
```

- Optimistic locking:

```
int pthread_mutex_trylock( pthread_mutex_t *lock);
  \\ returns 0 upon success
  \\ returns EBUSY upon failure
```

Hidden Race Conditions

Simple counter manipulation: `cnt++;`

Race Condition ??

Hidden Race Conditions

Simple counter manipulation: `cnt++;`

Race Condition ??

Race Condition !!

Hidden Race Conditions

Simple counter manipulation `cnt++;`

in (Pseudo-) Assembler:

Thread 1:

```
load cnt -> reg
add reg, 1 -> reg
store reg -> cnt
```

Thread 2:

```
load cnt -> reg
add reg, 1 -> reg
store reg -> cnt
```

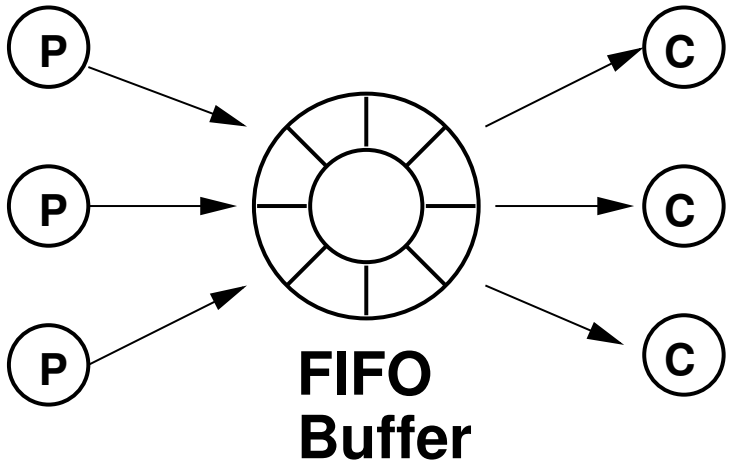
Problem:

- Old counter state first loaded into registered, then incremented and at last written back to memory without taking simultaneous icrements by other threads into account.

Multithreaded Programming with Posix Threads

Producer-Consumer Problems

Illustration:



Characteristics:

- Multiple producer processes / threads.
- Multiple consumer processes / threads.
- Single shared FIFO buffer.
- Producers write data to shared buffer.
- Consumers read data from shared buffer.

Example: Producer-Consumer Problem

Buffer data structure:

```
data buffer[BUFSIZE];
int occupied = 0;
int nextin = 0;
int nextout = 0;
pthread_mutex_t buflock = PTHREAD_MUTEX_INITIALIZER;
```

Example: Producer-Consumer Problem

Producer:

```
void Producer()
{
  data val;
  bool done;

  for (;;) {
    val = ... ;
    done = false;
    do {
      pthread_mutex_lock( &buflock);
      if (occupied < BUFSIZE) {
        buffer[ nextin] = val;
        nextin = (nextin + 1) % BUFSIZE;
        occupied = occupied + 1;
        done = true;
      }
      pthread_mutex_unlock( &buflock);
    } while (!done);
  }
}
```

Consumer:

```
void Consumer()
{
  data val;
  bool done;

  for (;;) {
    done = false;
    do {
      pthread_mutex_lock( &buflock);
      if (occupied > 0) {
        val = buffer[ nextout];
        nextout = (nextout + 1) % BUFSIZE;
        occupied = occupied - 1;
        done = true;
      }
      pthread_mutex_unlock( &buflock);
    } while (!done);
    DoSomething( val);
  }
}
```

Producer-Consumer Problem

Inefficiency of implementation:

- Producers and consumers synchronize even if they access disjoint buffer areas.
- Producer threads repeatedly acquire buffer lock although buffer is still full.
- Consumer threads repeatedly acquire buffer lock although buffer is still empty.

Producer-Consumer Problem

Inefficiency of implementation:

- Producers and consumers synchronize even if they access disjoint buffer areas.
- Producer threads repeatedly acquire buffer lock although buffer is still full.
- Consumer threads repeatedly acquire buffer lock although buffer is still empty.

Required mechanism:

- Suspend producer threads on full buffer.
- Wake-up producer threads after consumer threads have removed data items.
- Suspend consumer threads on empty buffer.
- Wake-up consumer threads after producer threads have provided data items.

Multithreaded Programming with Posix Threads

Condition Variables

Characteristics:

- CVs allow threads to suspend:
  - Producer suspends on full buffer.
  - Consumer suspends on empty buffer.

Condition Variables

Characteristics:

- CVs allow threads to suspend:
  - Producer suspends on full buffer.
  - Consumer suspends on empty buffer.
- CVs allow threads to trigger other threads:
  - Producer triggers suspended consumers.
  - Consumer triggers susepnded producers.

Condition Variables

Characteristics:

- CVs allow threads to suspend:
  - Producer suspends on full buffer.
  - Consumer suspends on empty buffer.
- CVs allow threads to trigger other threads:
  - Producer triggers suspended consumers.
  - Consumer triggers susepnded producers.
- CVs implement *event-driven critical regions*.

Condition Variables

Realization:

- Abstract data type:

```
pthread_cond_t
```

- Static initializer:

```
PTHREAD_COND_INITIALIZER
```

- Dynamic initialization:

```
int pthread_cond_init( pthread_cond_t      *cv,
                       pthread_condattr_t  *attr)
```

- Dynamic finalization:

```
int pthread_cond_destroy( pthread_cond_t  *cv)
```

Condition Variables

Suspend a thread:

- Suspend thread while waiting for condition.
- Unlock given mutex lock and re-acquire it prior to return.

```
int pthread_cond_wait( pthread_cond_t  *condvar,
                       pthread_mutex_t *lock)
```

Trigger a thread:

- Wake-up one thread suspended while waiting for condition.

```
int pthread_cond_signal( pthread_cond_t  *condvar)
```

- Wake-up all threads suspended while waiting for condition.

```
int pthread_cond_broadcast( pthread_cond_t  *condvar)
```

Producer-Consumer Problem with Condition Variables

Buffer data structure:

```
data buffer[BUFSIZE];
int occupied = 0;
int nextin = 0;
int nextout = 0;
pthread_mutex_t buflock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_cond_t space = PTHREAD_COND_INITIALIZER;
```

Producer-Consumer Problem with Condition Variables

Producer:

```
void Producer()
{
  for (;;) {
    val = ... ;

    pthread_mutex_lock( &buflock);
    while ( ! occupied < BUFSIZE) {
      pthread_cond_wait( &space,
                         &buflock);
    }
    /*  occupied < BUFSIZE */

    buffer[ nextin] = val;
    nextin = (nextin + 1) % BUFSIZE;
    occupied = occupied + 1;

    pthread_cond_signal( &items);
    pthread_mutex_unlock( &buflock);
  }
}
```

Consumer:

```
void Consumer()
{
  for (;;) {
    pthread_mutex_lock( &buflock);
    while ( ! occupied > 0) {
      pthread_cond_wait( &items,
                         &buflock);
    }
    /*  occupied > 0 */

    val = buffer[ nextout];
    nextout = (nextout + 1) % BUFSIZE;
    occupied = occupied - 1;

    pthread_cond_signal( &space);
    pthread_mutex_unlock( &buflock);

    DoSomething( val);
  }
}
```

The End: Questions ?