Beware of some German slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 5a: Kernel Level Threads in ULIX

Felix C. Freiling

Lehrstuhl für Praktische Informatik 1

Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

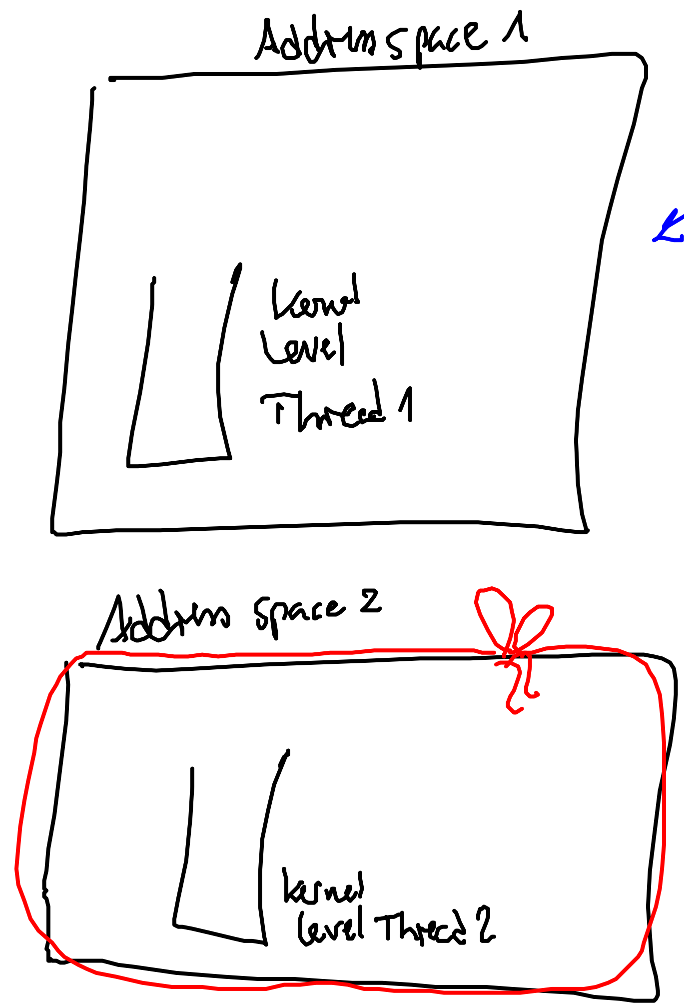
Overview

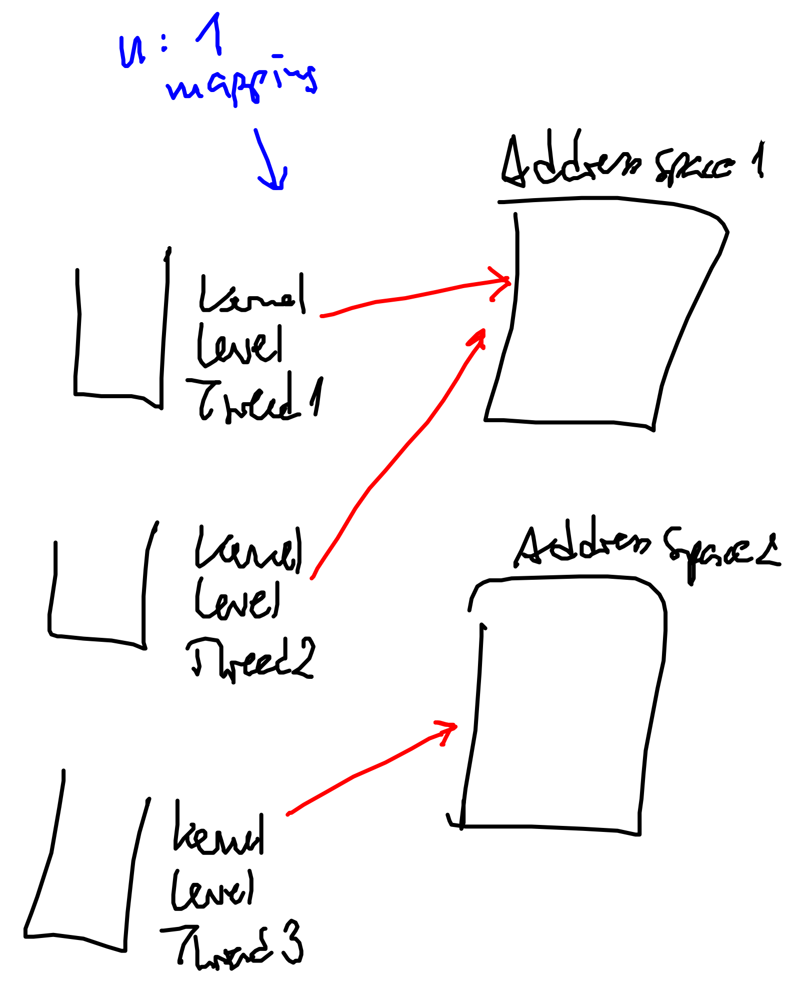
- Introduction
- Kernel Data Structures
- Handling Thread Queues
- Dispatcher Operations

Introduction

old world new world

Processes vs. Threads





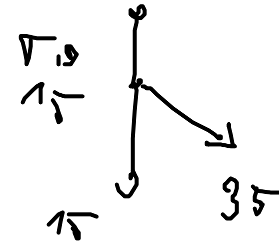
Threads in ULIX

- ULIX should support kernel level threads and user level threads
  - Kernel level threads are more basic, need to be implemented first
  - ULIX should support system calls like fork and exec
- ULIX should allow multiple kernel level threads in one virtual address space
  - System should support system call like klt_fork ...

klt_fork

- Like fork, only in same address space
- Result is another kernel level thread within the same address space
  - thread runs same user program as original thread
  - has its own user stack
  - klt_fork returns 0 in “child thread”
  - Child thread can them jump to its own program

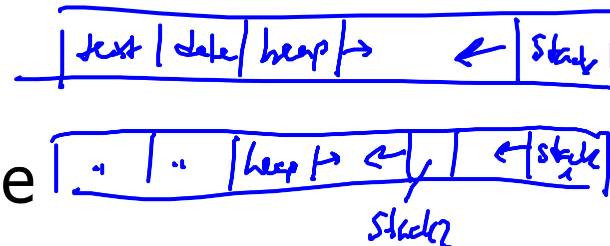
Example Program



```
void foo() { ... }
void bar() { ... }
main () {
  int p = klt_fork(); // start new KLT
  if (p == 0) {
    foo();  // child executes foo
  } else {
    bar();  // parent executes bar
  }
} // all in same virtual memory
```

fork() vs. klt_fork()

- fork() classically "only" creates new thread in a new address space
  - Runs same user program as original
  - Uses copy-on-write for performance
- klt_fork() does the same, only without creating new address space
  - Allocate new user stack
  - Create new TCB in thread table
  - Add new thread to ready queue

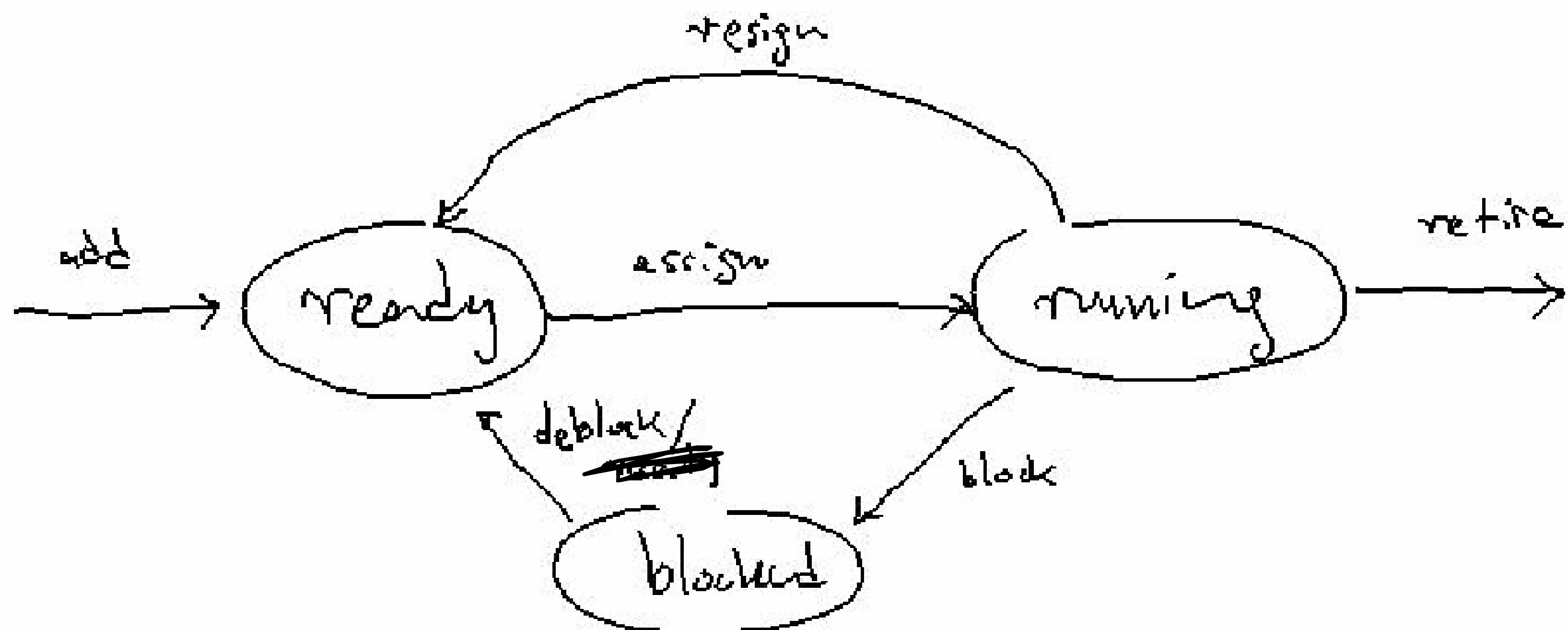


Kernel Data Structures

Thread State

⟨*kernel declarations* 34a⟩+≡

```
enum thread_state_enum { ready, running, blocked };
```



Thread Control Block

⟨*kernel declarations* 34a⟩+≡ (14b) ◁124a 124d▷

```
struct TCB {
  thread_id tid;
  unsigned int context[NUM_REGISTERS]; // general purpose registers
  unsigned int usp; // user stack pointer
  unsigned int pc; // program counter
  addr_space_id addr_space;
  ⟨more TCB entries 127⟩
};
```

Thread ID and Thread Table

$\langle$*kernel declarations* 34a$\rangle$+$\equiv$

```
#define MAX_THREADS 1024
```

$\langle$*kernel declarations* 34a$\rangle$+$\equiv$

```
TCB thread_table[MAX_THREADS];
```

$\langle$*kernel declarations* 34a$\rangle$+$\equiv$

```
typedef thread_id unsigned int;
```

- Thread ID is an index into thread table

Running Thread

$\langle$*kernel global variables* 108c$\rangle$+$\equiv$

```
thread_id[NUM_CPUS] running;
```

- Ready for multiprocessing
  - running[0] = id of thread running on cpu0

Handling Thread Queues

Pointers in the TCB

- The TCB has additional prev and next entries
  - Point to previous and next TCBs in "current list"

⟨*more TCB entries* 127⟩≡

```
thread_id next; // id of the ``next'' thread
thread_id prev; // id of the ``previous'' thread
```

- Results in double linked lists
  - Classical data structure
- Value 0 is "end" marker
  - There is no thread with ID 0

Ready Queue

- Since there is no thread with ID 0, we use TCB 0 for the ready queue
- Beginning of ready queue:

```
thread_table[0].next
```

- Last element of ready queue:

```
thread_table[0].prev
```

Blocked Queues

- For every type of event for which a thread may wait we need a separate blocked queue
- Blocked queues consist of a header element (like TCB 0 for ready queue):

```
⟨kernel declarations 34a⟩+≡
  struct blocked_queue {
    thread_id next; // id of the ``next'' thread
    thread_id prev; // id of the ``previous'' thread
  };
```

- Rest of blocked queues is implemented through pointer structures in TCBs

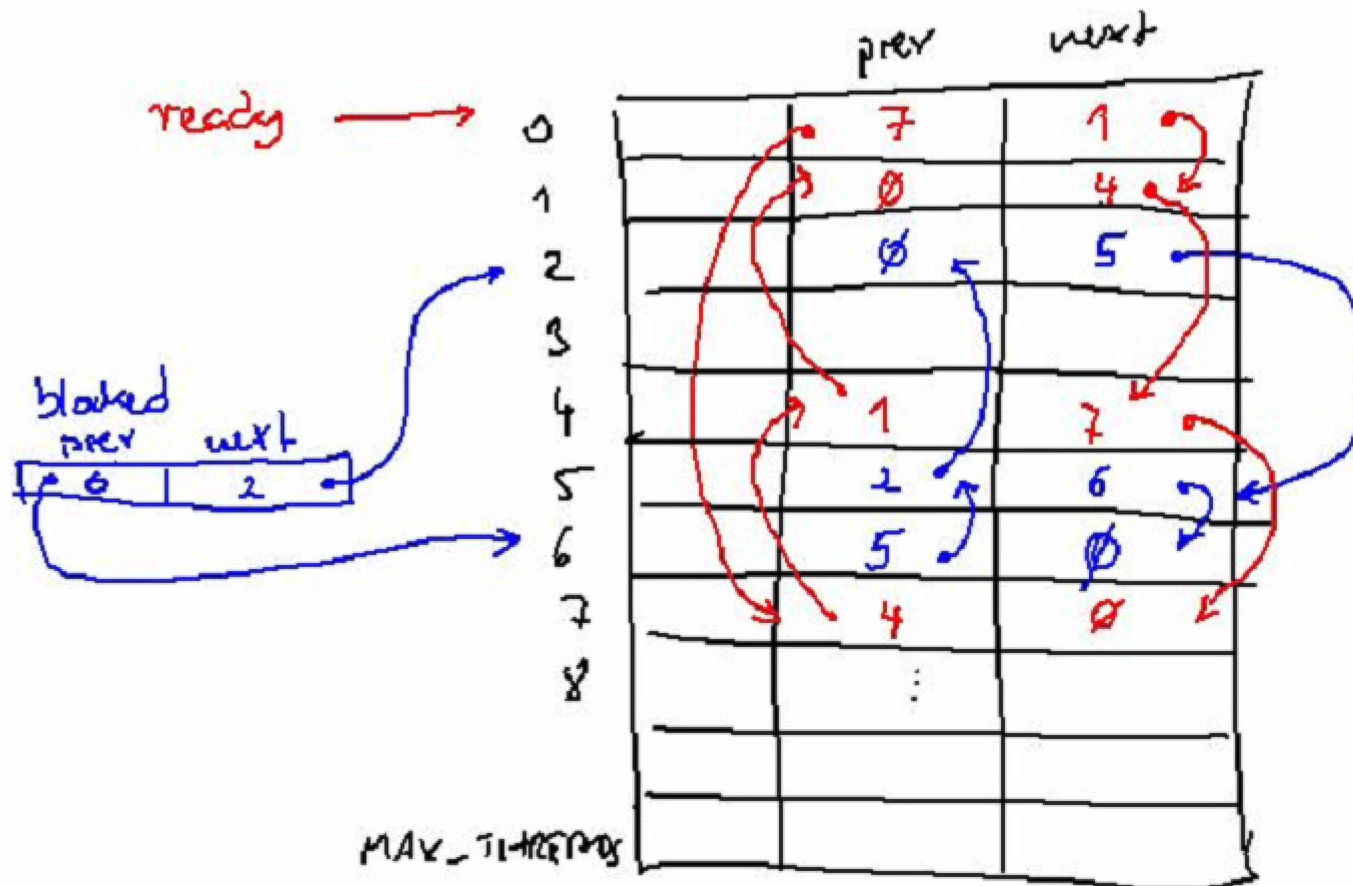


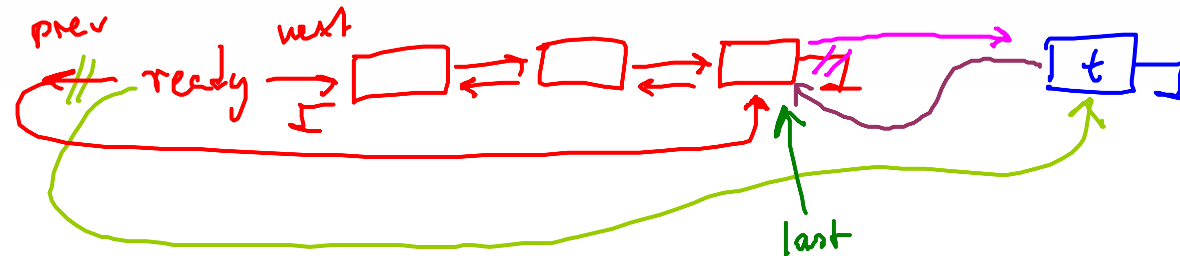
Figure 5.3: Implementation of ready queue and blocked queues. The beginning of the ready queue is implicitly defined by entry 0 in the thread table. The beginning of a blocked queue is a pair of thread identifiers pointing into the thread table from “outside”.

Adding to Ready Queue

- Adds to the end of ready queue

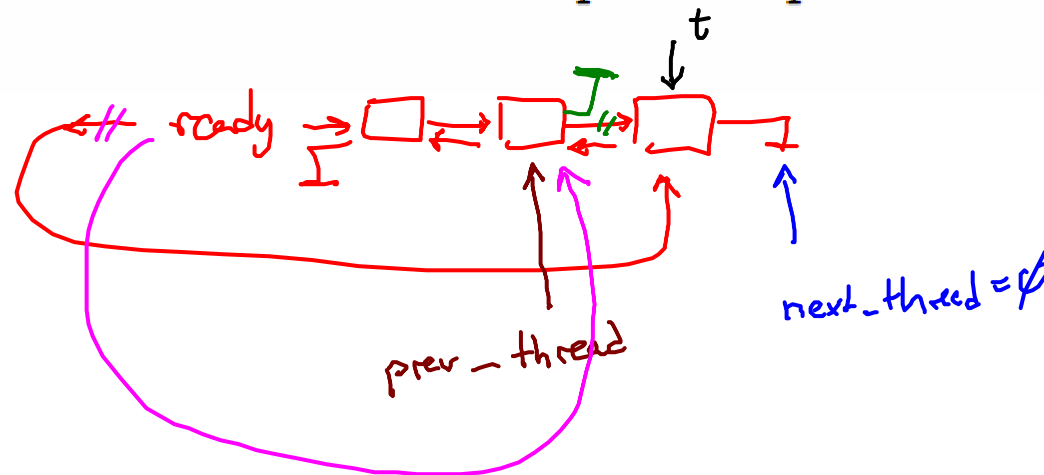
⟨*kernel functions* 110a⟩+≡

```
void add_to_ready_queue(thread_id t) {
    thread_id last = thread_table[0].prev;
    thread_table[0].prev = t;
    thread_table[t].next = 0;
    thread_table[t].prev = last;
    thread_table[last].next = t;
}
```



Removing from Ready Queue

```
⟨kernel functions 110a⟩+≡
  void remove_from_ready_queue(thread_id t) {
    thread_id prev_thread = thread_table[t].prev;
    thread_id next_thread = thread_table[t].next;
    thread_table[prev_thread].next = next_thread;
    thread_table[next_thread].prev = previous_thread;
  }
```



Adding to a Blocked Queue

⟨*kernel functions* 110a⟩+≡ (14b)

```
void add_to_blocked_queue(thread_id t, blocked_queue* bq) {
  thread_id last = bq.prev;
  bq.prev = t;
  thread_table[t].next = 0; // [[t]] is ``last'' thread
  thread_table[t].prev = last;
  if (last == 0) {
    bq.next = t;
  } else {
    thread_table[last].next = t;
  }
}
```

Removing from Blocked Queue

⟨*kernel functions* 110a⟩+≡ (14b) ◁130c

```
void remove_from_blocked_queue(thread_id t, blocked_queue* bq) {
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  if (prev_thread == 0) {
    bq.next = next_thread;
  } else {
    thread_table[prev_thread].next = next_thread;
  }
  if (next_thread == 0) {
    bq.prev = prev_thread;
  } else {
    thread_table[next_thread].prev = prev_thread;
  }
}
```

Front of Blocked Queue

- Given ...

```
⟨kernel functions 110a⟩+≡
  thread_id front_of_blocked_queue(blocked_queue bq) {
    return bq.next;
  }
```

- we can write for example:

```
deblock(front_of_blocked_queue(bq), &bq);
```

Dispatcher Operations

Dispatcher Operations

⟨*kernel declarations* 34a⟩+≡

```
void add(thread_id t);
void assign();
void block(blocked_queue* q);
void deblock(thread_id t, blocked_queue* q);
void resign();
void retire(thread_id t);
```

Easy Ones

⟨*kernel functions* 110a⟩+≡

```
void add(thread_id t) {
  add_to_ready_queue(t);
}
void retire(thread_id t) {
  remove_from_ready_queue(t);
}
void deblock(thread_id t, thread_id* blocked_queue) {
  remove_from_blocked_queue(t, blocked_queue);
  add_to_ready_queue(t);
}
```

was ist, wenn t running war?!

Resign for CPU 0

```
⟨kernel functions 110a⟩+≡
  void resign() {
    ⟨save processor context of running thread 132b⟩
    add_to_ready_queue(running[0]);
  }
```

Saving CPU Context

- Has to be written in assembly language
  - Inline assembler in C
- Example (not yet using current ULIX hardware instructions):

```
⟨save processor context of running thread 132b⟩≡                    (132a 134)
  __inline__ SAVE_REGISTERS(thread_table[running].context);
  __inline__ STA USP, thread_table[running].usp;
```

Assign for CPU 0

```
⟨kernel functions 110a⟩+≡
  void assign() {
    running = schedule(); // invoke the scheduler
    ⟨load processor context of running thread 133b⟩
  }
```

- Loading processor context also uses inline assembler
  - ... and eventually should use current ULIX assembly language ...

```
⟨load processor context of running thread 133b⟩≡
  __inline__ LOAD_REGISTERS(thread_table[running].context);
  __inline__ LDA USP, thread_table[running].usp;
  __inline__ LDAA (SSP) thread_table[running].pc;
```

Block

inline int foo() {

}

```
⟨kernel functions 110a⟩+≡
  void block(blocked_queue* q) {
    ⟨save processor context of running thread 132b⟩
    add_to_blocked_queue(running[0], q);
  }
```

- Can nicely use noweb macro mechanism for "inline" function calls
  - Maybe use this also for dispatcher operations?!

The Scheduler

⟨*kernel declarations* 34a⟩+≡

```
thread_id schedule();
```

- Still empty ...
- Implementation idea:
  - Enqueue at end
  - Dequeue at front
- Gives simple FIFO scheduler
- Need to add special thread in case ready queue is empty

Outlook

- Kernel level threads only 30-40% implemented
- User level threads 0% implemented (not too bad)