Beware of some German slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 6a:
Hardware and Thread Synchronization in ULIX

Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

Overview

- Critical sections and mutual exclusion
- Hardware Synchronization and Spin Locks
- Semaphores
- Implementation of Semaphores
- Synchronizing the Kernel
- User Level Semaphores

Critical Sections

- **Critical section:** a sequence of instructions of a program that accesses shared resources
- **Mutual exclusion**: at any time there is at most one thread in its critical section
- Critical sections are preceded by an **entry protocol** and succeeded by an **exit protocol**
- Notation (MUTEX = shorthand for mutual exclusion):
  - Entry protocol: ENTER_MUTEX
  - Exit protocol: EXIT_MUTEX

Three Abstraction Levels

- Entry and exit protocol are implemented differently on different levels of abstraction
  - Hardware level
  - Kernel level
  - User level
- We start with hardware level

Hardware Synchronization

- Synchronization at the lowest (hardware) level
  - Interrupt masking
  - For multiprocessors, additionally spin lock
- For spin lock we need a special hardware instruction
- In general:

```
ENTER_MUTEX =
  <disable interrupts>
  <spin lock>
EXIT_MUTEX =
  <give back lock>
  <enable interrupts>
```

Hardware Synchronization in ULIX

number	class/example
0	none
1	TRAP
2	timer interrupt
3	I/O interrupt
4	MMU interrupt (page fault)
5	division by zero (non-maskable)
6	basic protection violation (non-maskable)
7	invalid machine instruction encoding (non-maskable)

Table 2.1: Interrupt levels of the ULIX hardware.

- Disabling interrupts is done by interrupt masking
  - IER register in CPU, stores the highest allowed interrupt level
  - Interrupts above and including level 5 cannot be masked
- Disable interrupts could be implemented as

```
move byte IER, #7
```

Enabling Interrupts

- When we re-enable interrupts, which value should we assign IER?
- We need to remember previous interrupt level

```
<disable interrupts>=
  push byte IER    // push IER to system stack
  move byte IER, #7
<enable interrupts>=
  pop byte IER
```

- Does an interrupt between push IER and move do any harm?

Spin Locks

- Use special instruction SWAP of ULIX CPU
  - Example: swap int r0, r1
- Use global flag at symbolic address FLAG
  - Value 0: lock is free
  - Value 1: lock is taken

```
<spin lock>=
  SPIN: swap byte r0, FLAG
        jnz SPIN

<give back lock>=
        move byte FLAG, #0
```

Semaphore Semantics

- Assume semaphore S is initialized with k
- Then operations **P** and **V** on S have the following meaning:
  - P blocks in case exactly k threads have passed P without passing V
  - V deblocks a thread which is blocked at P in case such a thread exists
- For k=1, P and V can be used to implement ENTER_MUTEX and EXIT_MUTEX at a certain level of abstraction
  - Semaphores "use" blocking instead of busy waiting

Semaphores at Hardware Level?

- Can we use semaphores to implement low level synchronization?
  - Instead of turning off interrupts and busy waiting?
- Semaphores themselves contain critical sections (as we will see)
  - These critical sections cannot be implemented with semaphores
- Hardware synchronization is the only form of synchronization that does not use/need critical sections itself
  - Used to bootstrap synchronization abstractions bottom up

Semaphores in ULIX

- ULIX offers kernel level semaphores and user level semaphores
  - Operations prefixed with "kl_..." and "ul_..."
- Both have a similar structure
  - Kernel level semaphores use hardware synchronization to implement their critical sections
  - User level semaphores use kernel level semaphores to implement their critical sections
- Look at kernel level semaphores first

Semaphore Structure

```
⟨kernel declarations 34a⟩+≡                                   (14b) ◁130d 146b▷
  struct kl_semaphore {
    int counter;
    blocked_queue bq;
    ⟨more kl_semaphore entries 147d⟩  // uninteresting implementation details
  }
```

```
⟨kernel declarations 34a⟩+≡
  typedef kl_semaphore_id int;
```

↑ unsigned

```
⟨kernel declarations 34a⟩+≡
  kl_semaphore_id new_kl_semaphore(int k);
  void release_kl_semaphore(semaphore_id s);
```

Operation P

```
⟨kernel functions 110a⟩+≡                                  (14b) ◁135
  void kl_P(kl_semaphore_id sid) {
    kl_semaphore sem = ⟨semaphore structure with identifier sid 148a⟩;
    sem.counter = sem.counter - 1;
    if (sem.counter < 0) {
      block(sem.bq);
      assign();
    }
  }
```

interrupt

T1 T2 T3

new_sem(x)

kl_P(x) kl_P(x)

semct-- semct--

block

block

Operation V

⟨*kernel functions* 110a⟩+≡ (14b) ◁146d

```
void kl_V(kl_semaphore_id sid) {
  kl_semaphore sem = ⟨semaphore structure with identifier sid 148a⟩;
  if (sem.counter < 0) {
    deblock(front_of_blocked_queue(bq), &bq);
  }
  sem.counter = sem.counter + 1;
}
```

Semaphore Table

⟨*kernel declarations* 34a⟩+≡ (14b) ◁146c 147c▷

```
kl_semaphore kl_semaphore_table[MAX_SEMAPHORES];
```

Uses MAX_SEMAPHORES 147c.

There's a maximum number of semaphores that can be allocated in the kernel.

⟨*kernel declarations* 34a⟩+≡ (14b) ◁147b

```
#define MAX_SEMAPHORES 32
```

Defines:
MAX_SEMAPHORES, used in chunks 147 and 148c.

Since both used and unused semaphores are held in a table, we need additional information to distinguish both. So each semaphore has a counter and a queue, but it also has an additional field storing the semaphore state. The value `false` means the semaphore entry is free.

⟨*more* kl_semaphore *entries* 147d⟩≡ (146a)

```
boolean used;
```

⟨*semaphore structure with identifier* sid 148a⟩≡

```
kl_semaphore_table[sid]
```

Getting a New Semaphore

⟨*kernel global variables* 108c⟩+≡

```
kl_semaphore_id next_kl_semaphore = 0;
```

⟨*kernel functions* 110a⟩+≡ (14b) ◁147a 148d▷

```
kl_semaphore_id new_kl_semaphore(int k) {
  int check = MAX_SEMAPHORES;
  while (kl_semaphore_table[next_kl_semaphore].used == true) {
    next_kl_semaphore = (next_kl_semaphore + 1) % MAX_SEMAPHORES;
    check = check - 1;
    if (check <= 0) {
      return -1;
    }
  }
  kl_semaphore_table[next_kl_semaphore].used = true;
  kl_semaphore_table[next_kl_semaphore].counter = k;
  initialize_blocked_queue(kl_semaphore_table[next_kl_semaphore].bq);
  return next_kl_semaphore;
}
```

Releasing a Semaphore

⟨*kernel functions* 110a⟩+≡ (14b) ◁

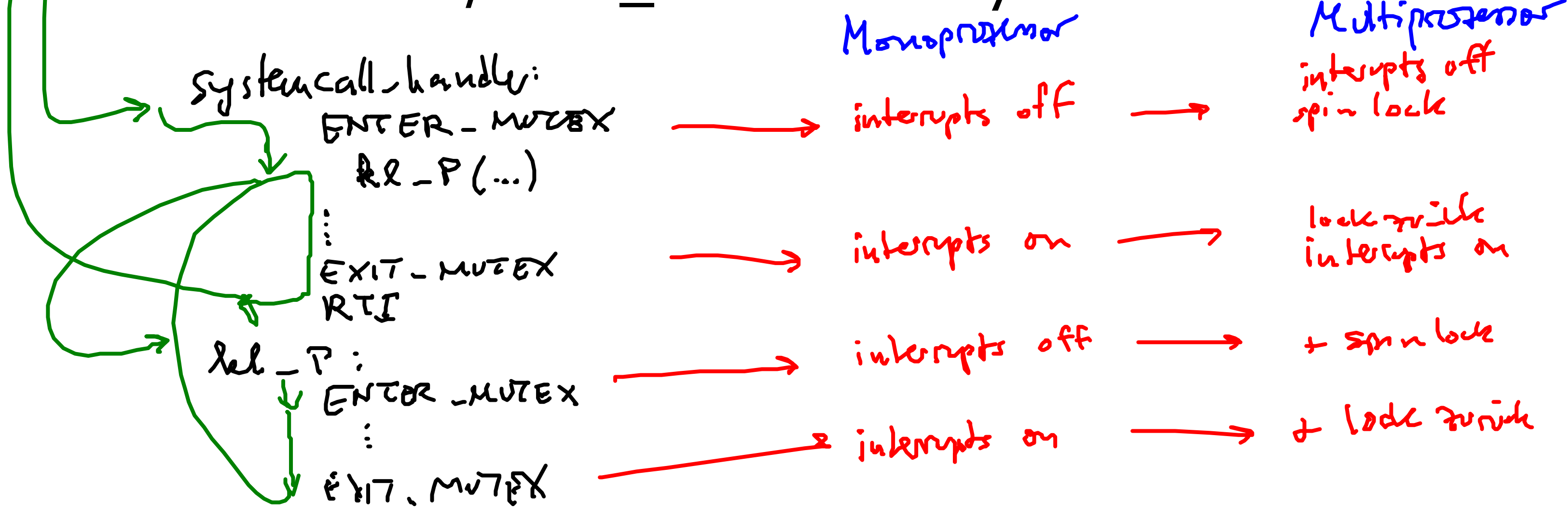
```
void release_kl_semaphore(semaphore_id s) {
  kl_semaphore_table[s].used = false;
  while (front_of_blocked_queue(kl_semaphore_table[s].bq) != 0) {
    thread_id t = front_of_blocked_queue(kl_semaphore_table[s].bq);
    remove_from_blocked_queue(t, kl_semaphore_table[s].bq]);
    add_to_ready_queue(t);
  }
}
```

Semaphores and Critical Sections

- Semaphores themselves contain critical sections
  - Two threads T1 and T2
  - Both invoke kl_P on same semaphore initialized with 1
  - T1 interrupted after decrementing counter (before checking)
  - T2 decrements and checks (blocks)
  - Control returns to T1
  - T1 also blocks (since counter is below 0)
- Critical sections should be declared using ENTER_MUTEX and EXIT_MUTEX
  - Implemented with hardware mechanisms

Semaphores in Context

- Semaphore operations are used in system calls
  - System calls (interrupt handlers) can also be regarded as critical sections
  - ENTER/EXIT_MUTEX everywhere



ENTER/EXIT_MUTEX Where?

- Critical sections should be declared in the kernel consistently
  - Either all system calls are critical sections or all calls from system calls are critical sections or ...
- Two possible forms:
  - **Strict** kernel synchronization: entire kernel is a (big) critical section
  - **Concurrent** kernel synchronization: only parts of kernel code are critical sections

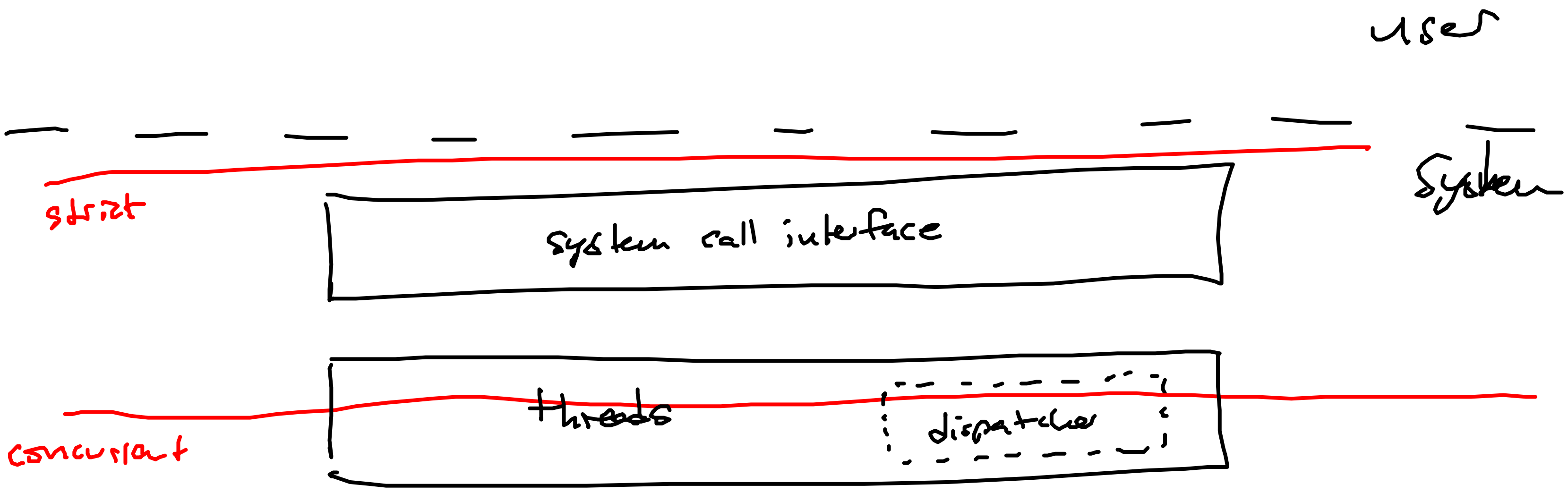
Strict Kernel Synchronization

- Critical section begins as soon as kernel mode is entered
  - Through a system call or asynchronous interrupts
- System calls always run to completion (are not interrupted)
- On multiprocessors, only one CPU can be in kernel mode at the same time
- Conceptual simplicity:
  - Mutual exclusion achieved (easy to see and enforce)
- Not very efficient

Concurrent Kernel Synchronization

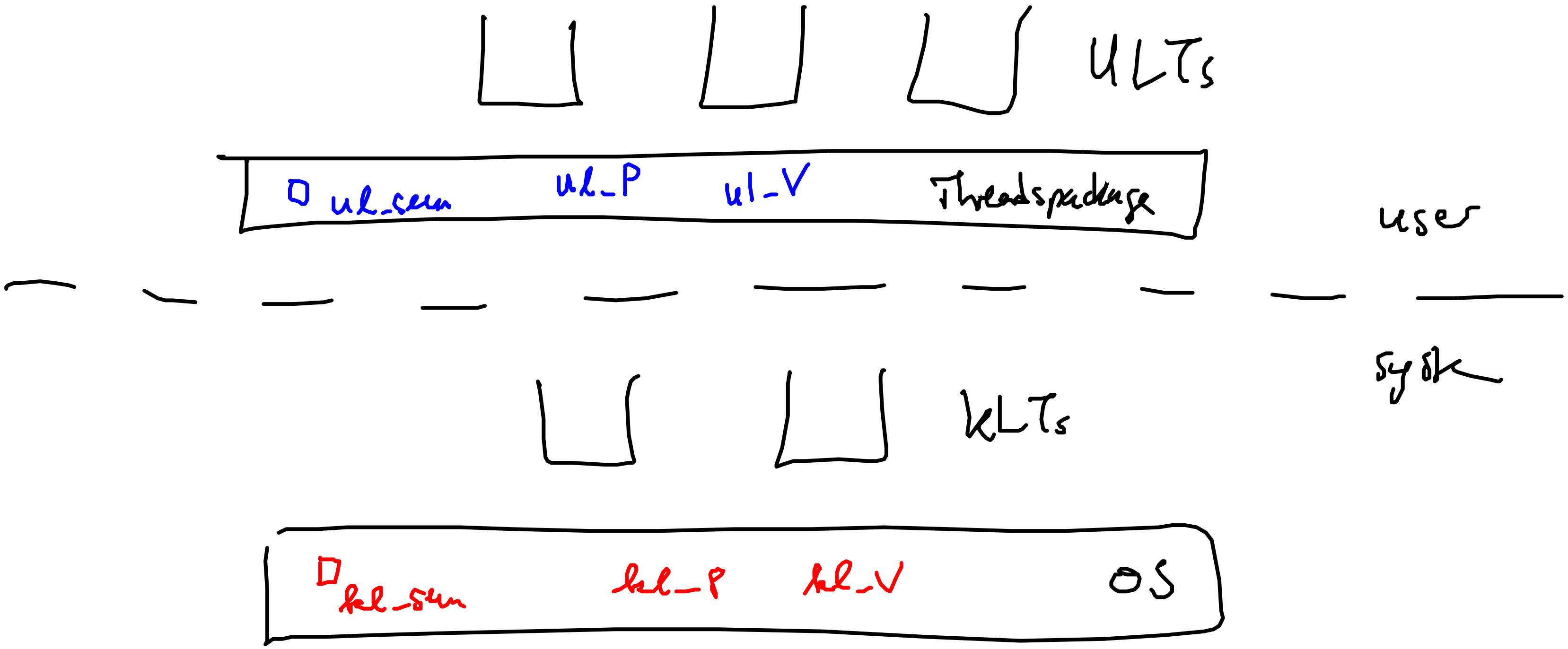
- Critical sections should be as short as possible to enable concurrency
- Only declare those parts of the code as critical sections that access shared data structures
- Example: only functions on the level of the dispatcher are critical sections
- Much more efficient, but
  - much harder to program correctly
  - system stack can become pretty messy

Strict vs. Concurrent Synchronization



User Level Semaphores

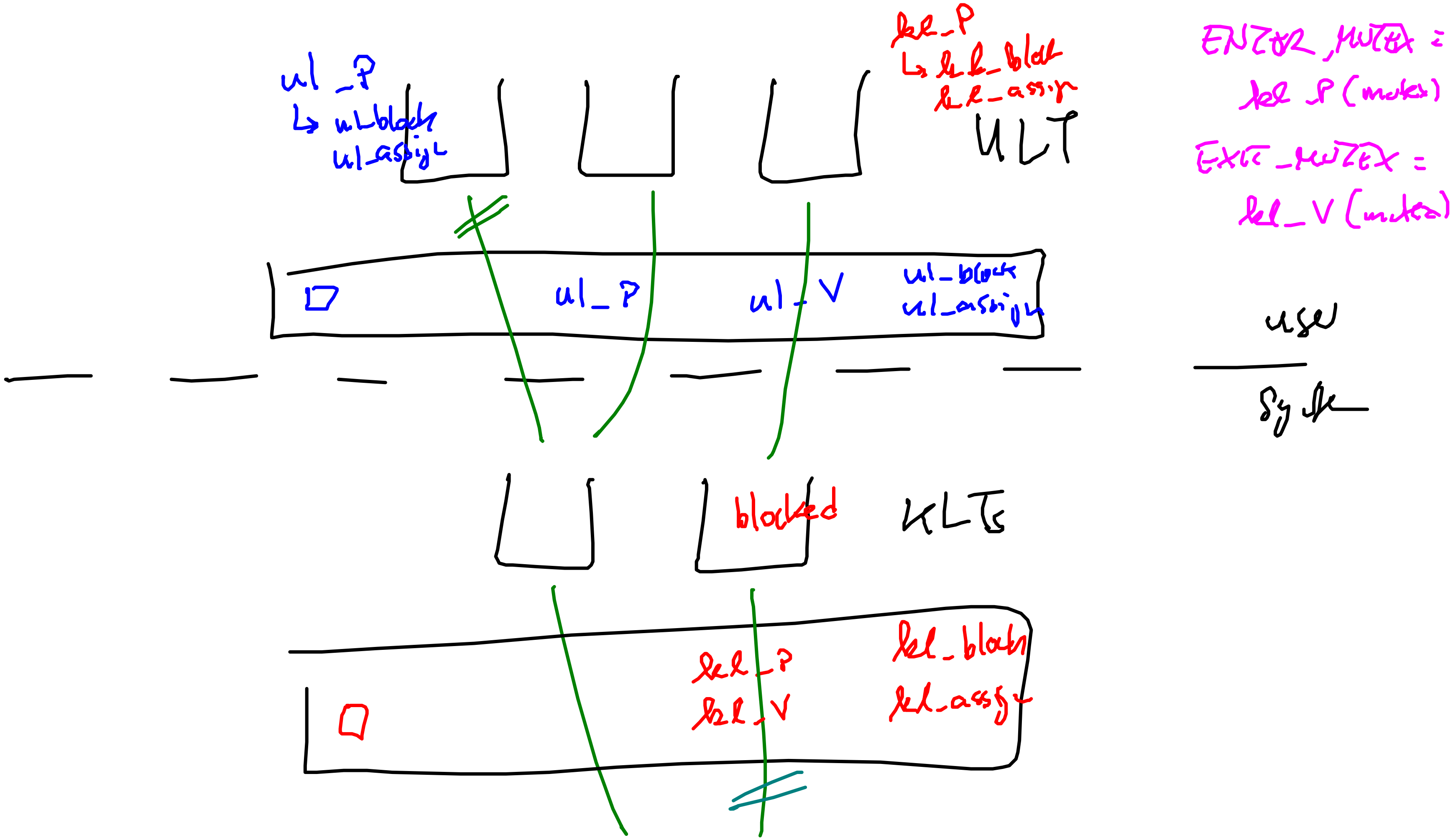
- Implemented for user level threads (in the threadspackage)



Implementation

- Implementation is copied from kernel level
  - Structure containing counter and a blocked queue (of user level threads)
- Operation P:
  - Check counter
  - If below 0, block on queue and assign
- Operation V
  - Deblock and increase counter
- Use dispatcher operations of threads package!

Kernel vs. User Level



Synchronization Hierarchy

- High level: user level semaphores
  - Uses middle level as implementation
- Middle level: kernel level semaphores
  - Uses low level as implementation
- Low level: interrupt masking and spin locks

Summary

- Critical sections and mutual exclusion
- Hardware Synchronization and Spin Locks
- Semaphores
- Implementation of Semaphores
- Synchronizing the Kernel
- User Level Semaphores