Locks (ch. 28) pt. 1

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Kernel mode, Single processor

Evaluating Locks

- **Mutual exclusion**
  - At most one thread in the CS
- **Deadlock-freedom**
  - Some thread eventually enters CS
- **Fairness** (**starvation-freedom**)
  - Each thread eventually enters CS
- **Performance**
  - Time overhead for using the lock
  - Single thread: overhead for grab & release
  - Multiple threads and CPUs

Get Free Page

```c
struct run {
  struct run *next;
};
struct run *freelist;

void *kalloc(void)
{
  struct run *r;

  r = freelist;
  if (r)
     freelist = r->next;

  return (void*)r;
}
```

Single processor

Recall

- The kernel is a multi threaded app.
- At minimum each process is a thread in the kernel.
- If there are user mode threads, there might be more kernel threads.
- Context switch happens in the kernel.

So. What is the problem with the kalloc?

Single processor

Recall

- The kernel is a multi threaded app.
- At minimum each process is a thread in the kernel.
- If there are user mode threads, there might be more kernel threads.
- Context switch happens in the kernel.

So. What is the problem with the kalloc?
Race condition.
Solution?

Single processor

Recall

- The kernel is a multi threaded app.
- At minimum each process is a thread in the kernel.
- If there are user mode threads, there might be more kernel threads.
- Context switch happens in the kernel.

So. What is the problem with the kalloc?
Race condition.
Solution?
Critical section.

Single processor, Critical section

```c
void *kalloc(void)
{
  struct run *r;

  intDsbl();

  r = freelist;
  if (r)
     freelist = r->next;

  intEnbl();

  return (void*)r;
}
```

Is this good?

- It works.
- The interrupts are disabled for a SHORT period of time.

So, it is reasonable. No better solution.
However. What happens if the critical section is one of the following:

- long
- requires interrupt machinery (which anyways means long)

softLock

```
int mutex = 0;

void lock() {
    if (mutex == 0) {
        mutex = 1;
        return;
    };
    //
    // How do we wait for mutex == 0?
    //
}
```

softLock

```
int mutex = 0;

void lock() {
    if (mutex == 0) {
        mutex = 1;
        return;
    };
    while (mutex == 0);
    mutex = 1;
}
```

Problems

- Ugly code.
- Race conditions.
- Let us add intDsbl/intEnbl

softLock

```
int mutex = 0;

void lock() {
    intDsbl();
    if (mutex == 0) {
        mutex = 1;
        intEnbl();
        return;
    };
    while (mutex == 0);
    mutex = 1;
    intEnbl();
}
```

softLock DEADLOK

- If we get into the busy wait, we are stuck forever.
- We are in kernel. We can BLOCK.

BLOCK

- BLOCKs (i.e., set state to blocked) the kernel thread.

```
      proc->state = BLOCKED;
//
      sched();
//
```

- Note that intDsbl is in on process and intEnbl in the following one.

softLock

```c
int mutex = 0;

void lock() {
  intDsbl();
  if (mutex == 0) {
    mutex = 1;
    intEnbl();
    return;
  };
  BLOCK();
  mutex=1;  //One UNBLOCKed, OK. Else, bad.
  intEnbl();
}
```

softLock, good one

```c
int mutex = 0;

void lock() {
  intDsbl();
  while (mutex == 1) BLOCK();
  mutex = 1;
  intEnbl();
  return;
}
```

UNBLOCK

- Coarse: UNBLOCK moves ALL blocked kernel threads (i.e., processes) to ready.

```
    For each process if state BLOCK then
        set to READY
```

- Fine: UNBLOCK(pid)

softUnlock

```
int mutex = 0;

void unlock() {
  intDsbl();
  mutex = 0;
  UNBLOCK();
  intEnbl();
}
```

note

- UNBLOCK might be too long.
- The intDsbl/intEnbl create a critical section in the soft lock implementation.
- This is a very coarse lock.
- We can use many mutexes in order to get fine grain soft locks.
- We can use UNBLOCK which wakes up only one thread waiting for the specific mutex.

Kernel mode, Multiple processors

The race condition is BACK!

```c
void *kalloc(void)
{
  struct run *r;

  intDsbl();

  r = freelist;
  if (r)
     freelist = r->next;

  intEnbl();

  return (void*)r;
}
```

hardLock

```c
void hardLock(int *mutex) {
  intDsbl();
  while ( *mutex != 0 );
  mutex = 1;
  intEnbl();
}
```

Still race condition!!

Test-And-Set

Machine instruction:

TAS mem,new,reg

- Hardware support: a new instruction **test-and-set**
  - Update value and return previous, **atomically** across all processors
- Defined as:

```
int TestAndSet(int* mem, int new) {
    int old = *mem
    *mem = new;
    return old;
}
```

hardLock

```c
void hardLock(int *mutex) {
  intDsbl();
  while (tas(mutex, 1));
  intEnbl();
}

void hardUnlock(int *mutex) {
  *mutex = 0;
}
```

TAS not unique: Compare-And-Swap

Machine instruction:

CAS mem,expected,new,reg

- Another hardware primitive: **compare-and-swap**
- Compare to `expected`, update only if equal, return previous
- Defined as:

```
int CompareAndSwap(int* ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

hardLock (equivalent to the prev one)

```
void hardLock(int *mutex) {
  intDsbl();
  while (cas(mutex, 0, 1));
  intEnbl();
}

void hardUnlock(int *mutex) {
  *mutex = 0;
}
```

Decker's Algorithm: Without hardware support

- What about a lock without hardware support?

```
int flag[2];     // wants to grab lock?
int turn;        // whose turn?

void init() {
    flag[0] = flag[1] = 0;
    turn = 0;
}
void lock(int self) {
    flag[self] = 1;
    turn = 1 - self;    // let other run
    while ((flag[1-self] == 1) && (turn == 1-self))
        ; // spin-wait
}
void unlock(int self) {
    flag[self] = 0;
}
```

- Various issues → concurrency course

kalloc, good one

```
struct {
  struct run *freelist;
  int mutex;
} kmem;

void *kalloc(void)
{
  struct run *r;

  hardLock(&kmem.mutex);
   r = kmem.freelist;
  if (r)
     kmem.freelist = r->next;
   hardUnlock(&kmem.mutex);
   return (void*)r;
}
```

Again. Is this good?

- It works.
- The interrupts are disabled for a SHORT period of time.
- The busy-wait (aka spin-lock) will run for a SHORT period of time.

So, it is reasonable. No better solution.
However. What happens if the critical section is one of the following:

- long
- required interrupt machinery (which anyways means long)

Easy now. Use hardLock/hardUnlock instead of intDsbl/intEnbl.

softLock

```
struct {
  int haMutex;
  int mutex;
} softMutex;

softLock(softMutex *sMutex) {
    hardLock(&sMutex->haMutex);

    while (s->mutex != 0) {
        hardUnlock(&sMutex->haMutex);
        BLOCK();
        hardLock(&sMutex->haMutex);
    }
    sMutex->mutex = 1;

    hardUnlock(&sMutex->haMutex);
}
```