

Lab 3 — Synchronization

Background

- Process can execute concurrently or in parallel.
- Coordinating access to shared resources
- Problems
 - Race conditions
 - Deadlocks
 - Resource starvation
- Solutions
 - Synchronization: locks, barriers, semaphores, etc.
 - ...

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has a **critical section** segment of code that is shared with at least one other process
 - Process may be changing common variables, updating the table, writing a file, etc
 - When one process is in the critical section, no other may be in its critical section
- **Critical section problem** is to design a protocol that the process can use to synchronize their activity to cooperatively share data.

Critical Section

- The section of the code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, the remaining code is the **remainder section**

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

General structure of process P_i

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical-Section Handling in OS

Two approaches, depending on if the kernel is preemptive or non-preemptive

- **Preemptive** – allows a process to be preempted while it is running in kernel mode.
- **Non-preemptive** – does not allow a process running in kernel mode to be preempted, a kernel mode process will run until it exists kernel mode, blocks, or voluntarily yields control of the CPU.
 - A non-preemptive kernel is essentially free of race conditions in kernel mode

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- It is restricted to **two processes** solution that alternate **execution** between their **critical sections** and remainder sections.
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- It requires the two processes to share two variables:
 - `int turn;`
 - `boolean flag[2]`
- The variable `turn` indicates whose **turn it is to enter the critical section**, If it is 1 then the process is allowed to execute in its critical section.
- The `flag` array is used to indicate **if a process is ready to enter the critical section**. `flag[i] = true` implies that process P_i is ready!

Synchronization Hardware

- Up to now, software-based solution to critical-section problem which do not guarantee to work on modern computer architecture.
- Three hardware instructions that provide support for solving the critical-section problem:
 - Memory barriers
 - Hardware instructions
 - Atomic variables

Race Condition: quando siamo in una situazione di concorrenza o accesso a dati e risorse condivisi, ci troviamo di fronte ad un problema di tipo Race Condition — cioè il risultato finale potrebbe dipendere dall'ordine con cui i vari task sono eseguiti.

Noi vorremmo che il risultato non dipendesse dall'ordine con cui eseguiamo i task.

Deadlocks: A è in attesa che B finisca, il quale è in attesa che C finisca, il quale è in attesa che A finisca. Attesa circolare.

Resource Starvation: alcuni task non riescono ad ottenere le risorse necessarie per continuare l'esecuzione, perchè occupate indefinitivamente da altri processi.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problems.
- Simplest is **Mutex Lock** for protecting critical sections and preventing race condition:
 - A process must acquire a lock before entering a critical section using `acquire()` function
 - Releasing the lock when it exits the critical section using the `release()` function
 - A mutex lock has a Boolean variable available whose value indicates if the lock is available or not.
 - If the lock is available, a call to `acquire()` succeeds and the lock is considered unavailable.

Mutex Locks

- Calls to `acquire()` and `release()` must be atomic.
 - Usually implemented via hardware atomic instructions such as compare-and-swap.
 - **Disadvantage:** this solution requires **busy waiting**, while the process is in the critical section, the other processes must continuously loop to `acquire()`.
 - Wasting CPU cycle.
- This lock is therefore called a **spinlock**.
 - **Advantage:** no context switch is required when a process might wait on a lock. If a lock is to be held for a short duration, one thread can “spin” on one processing core while another thread performs its critical section on one another.

La soluzione implementata tramite Mutex Locks comporta un'attesa di tipo **busy waiting**, perchè un thread che cerca di acquisire il lock per entrare in sezione critica continua a chiamare `acquire()` finchè non riesce ad acquisire il lock. Il vantaggio è che, però, se il lock è acquisito da un altro thread per un breve periodo, il thread che continua a chiamare `acquire()` in esecuzione su un altro core, potrà entrare presto in sezione critica senza alcun bisogno di context switch.

Solution to Critical-section Problem Using Locks

```
while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}
```

What Should you do if you can't get a lock?

- Keep trying
 - “spin” or “busy-wait”
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor

Differenza tra busy waiting e non-busy waiting:

- busy waiting: chiedo in continuazione se il lock è occupato, e quando si libera lo acquisisco
- non-busy waiting: chiedo se il lock è occupato — se lo è, vado a “dormire” sapendo che verrò svegliato da qualcun altro quando il lock sarà libero, in modo da poterlo acquisire

Per implementare entrambi i tipi di attesa, abbiamo bisogno di supporto per le operazioni **atomiche**.

Lock implementation

- Need hardware support
- Uniprocessors – could disable interrupts
 - execute without preemption
 - Cannot be extended to multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
Atomic = non-interruptible
 - Either test memory word and set value (test-and-set)
 - Or swap contents of two memory words (compare-and-swap)



Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
 1. Memory barriers
 2. Hardware instructions 
 3. Atomic variables

Noi vedremo in particolare il meccanismo di **Hardware instructions**, ma vediamo brevemente anche le altre.



Memory Barriers

- Memory model are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
 - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors.
 - Weakly ordered – where a memory modification of one processor may not be immediately visible to all other processors.
 
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory barriers

Le letture e scritture in memoria devono essere consistenti: se un processore scrive una cella di memoria, un altro processore non può leggere tale cella di memoria finché il primo processore non ha finito la scrittura.



Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction



test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to `true`



Mutex Lock Definitions

```
■ acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

■ release() {
    available = true;
}
```

These two functions must be implemented atomically.
Both test-and-set and compare-and-swap can be used to implement these functions.

L'implementazione di un Mutex Lock fondamentalmente prevede il seguente comportamento: *finchè il lock non è disponibile faccio busy waiting. Non appena diventa disponibile, lo acquisisco settandolo nuovamente "non disponibile"* (così che altri threads non possano entrare in sezione critica) *ed entro in sezione critica. Una volta finito, faccio la release(): rendo cioè disponibile il lock agli altri threads.*

Il problema sta nell'acquire(): supponiamo che 2 processi, che contemporaneamente sono in busy waiting, ad un certo punto leggono che il lock è disponibile (available == true). Entrambi quindi entrano in sezione critica e settano available = false, ciascuno pensando di essere il proprietario del lock. Questo è un problema.

La soluzione consiste nel fare in modo che

non ci sia alcun tempo tra il momento in cui leggo available == true e setto available = false.

Abbiamo cioè bisogno di un'operazione **atomica**: test_and_set().



Solution using test_and_set()

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {
    while(test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

`lock == 0` → available

`lock == 1` → not available

Finchè `test_and_set(&lock)` ritorna `true` (dunque `lock` non available) → busy waiting.

Non appena torna

`false`, lo ri-setta a `true` atomicamente, ma consentendo al thread di entrare in sezione critica, mentre tutti gli altri leggeranno `true` (`lock` non disponibile).

Questa operazione atomica di test and set può essere svolta solo da un thread alla

volta, poichè atomica, quindi non c'è possibilità che 2 threads entrino in sezione critica contemporaneamente.



Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

Simile al comportamento di `test_and_set()`, ma invece di basarsi su un boolean (true/false), si basa su una coppia di variabili.



Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment	atomic_int *v)
{
    int temp;      /*

    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v, temp, temp+1)));
}
```

Spinlock

- “spin” or “busy-wait” 
- Good if delays are short
 - Fast critical section
- Can be implemented by in many ways
 - Test-and-set
 - Optimized (for performance) versions: test + test-and-set.
 - Loop on test (no bus contention)
 - When (possibly) free do test-and-set (with bus contention)

Mutual Exclusion Using a Semaphore



```
struct semaphore *s;  
s = sem_create("MySem1", 1); /* initial value is 1 */  
  
P(s); /* do this before entering critical section */  
  
critical section /* e.g., call to list remove front */  
  
V(s); /* do this after leaving critical section */
```



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **s** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - ▶ (Originally called **P()** and **V()**)

- Definition of the **wait()** operation

```
wait(s) {  
    while (s <= 0)  
        ; // busy wait  
    s--;  
}
```

- Definition of the **signal()** operation

```
signal(s) {  
    s++;  
}
```



Il semaforo può essere visto come **una estensione di un lock**, nel senso che un semaforo binario è essenzialmente come un lock.

Il semaforo però non è usato solo come meccanismo di mutua esclusione, ma anche come meccanismo di sincronizzazione tra due processi.

OS/161: Disabling Interrupts

- On a uniprocessor, only one thread at a time is actually running.
- If the running thread is executing a critical section, mutual exclusion may be violated if
 1. the running thread is preempted (or voluntarily yields) while it is in the critical section, and
 2. the scheduler chooses a different thread to run, and this new thread enters the same critical section that the preempted thread was in
- Since preemption is caused by timer interrupts, mutual exclusion can be enforced by disabling timer interrupts before a thread enters the critical section, and re-enabling them when the thread leaves the critical section. This is the way that the OS/161 kernel enforces mutual exclusion. There is a simple interface (`splhigh()`, `spl0()`, `splx()`) for disabling and enabling interrupts. See `kern/arch/mips/include/spl.h`.

Come possiamo implementare un'attesa di tipo *non-busy waiting*?

Su un sistema single core ciò può essere fatto disabilitando gli interrupt, sapendo che solo un thread alla volta è in esecuzione sul processore.

Quindi quando il thread entra in sezione critica, disabilitiamo gli interrupt, in modo tale che altri thread non possano interrompere l'esecuzione per venire schedulati e possibilmente entrare anch'essi in sezione critica.

Su un sistema multi core questa strategia non funziona.

In questo contesto abbiamo bisogno di primitive di sincronizzazione.



Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {           ⌂
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

modo di implementare un semaforo senza busy waiting.

Decremento il valore del semaforo: se dopo aver decrementato vedo che il valore è minore di zero, allora devo aspettare → block().

Qualcun altro mi sveglierà quando il valore sarà stato incrementato → wakeup(P).

OS/161 Locks

- OS/161 also uses a synchronization primitive called a *lock*. Locks are intended to be used to enforce mutual exclusion.



```
struct lock *mylock = lock create("LockName");  
  
lock_acquire(mylock);  
    critical section /* e.g., call to list remove head */  
lock_release(mylock);
```

- A **lock is similar to a binary semaphore** with an initial value of 1. However, locks also enforce an additional constraint: the thread that releases a lock must be the same thread that most recently acquired it.
- The system enforces this additional constraint to help ensure that locks are used as intended.

In OS161 troviamo già implementati gli spinlocks, e dobbiamo implementare i locks.



Problems with Semaphores

- Incorrect use of semaphore operations:
 - `signal (mutex) ... wait (mutex)`
 - `wait (mutex) ... wait (mutex)`
 - Omitting of `wait (mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

In OS161, in `kern/thread/thread.c`, troviamo già le implementazioni a basso livello delle primitive di wait e signal:

- `wchan_create`
- `wchan_destroy`
- `wchan_sleep`
- `wchan_wakeone`
- `wchan_wakeall`

Le prime due creano e distruggono una certa struttura dati chiamata ***wait channel***.

`wchan_sleep` → wait

`wchan_wakeone` → signal 1 sleeping process

`wchan_wakeall` → signal all sleeping process

Il file `kern/thread/synch.c` contiene le funzioni per la gestione dei semafori, locks e condition variables.

Semafori

I semafori sono già completamente implementati.

In

`kern/include/synch.h` troviamo la definizione della struttura di un semaforo:

```

40  /*
41   * Dijkstra-style semaphore.
42   *
43   * The name field is for easier debugging. A copy of the name is made
44   * internally.
45   */
46  struct semaphore {
47      char *sem_name;
48      struct wchan *sem_wchan;
49      struct spinlock sem_lock;
50      volatile unsigned sem_count;
51  };
52
53  struct semaphore *sem_create(const char *name, unsigned initial_count);
54  void sem_destroy(struct semaphore *);
55
56  /*
57   * Operations (both atomic):
58   *     P (proberen): decrement count. If the count is 0, block until
59   *                   the count is 1 again before decrementing.
60   *     V (verhogen): increment count.
61   */
62  void P(struct semaphore *);
63  void V(struct semaphore *);

```

Diamo un'occhiata al funzionamento dei semafori (synch.c).

```

45 // Semaphore.
46
47 struct semaphore *
48 sem_create(const char *name, unsigned initial_count)
49 {
50     struct semaphore *sem;
51
52     sem = kmalloc(sizeof(*sem));
53     if (sem == NULL) {
54         return NULL;
55     }
56
57     sem->sem_name = kstrdup(name);
58     if (sem->sem_name == NULL) {
59         kfree(sem);
60         return NULL;
61     }
62
63     sem->sem_wchan = wchan_create(sem->sem_name);
64     if (sem->sem_wchan == NULL) {
65         kfree(sem->sem_name);
66         kfree(sem);
67         return NULL;
68     }
69
70     spinlock_init(&sem->sem_lock);
71     sem->sem_count = initial_count;
72
73     return sem;
74 }
```

`sem_create` riceve come parametri il **nome** del semaforo, e il **counter** a cui deve essere inizializzato.

Vediamo che la funzione alloca una struttura di tipo `semaphore`, vi assegna il nome passato come parametro e vi assegna un wait channel, una struttura dati che viene allocata tramite

`wchan_create`.

Inoltre inizializza lo spinlock del semaforo.

Notiamo, che al contrario del wait channel, lo spinlock (nella definizione della struttura del semaforo) non è un puntatore, quindi non necessita di essere allocato, cosa che invece succede per il wait channel.

Viene anche inizializzato il counter del semaforo tramite il parametro passato alla funzione.

```
76 void
77 sem_destroy(struct semaphore *sem)
78 {
79     KASSERT(sem != NULL);
80
81     /* spinlock_cleanup will assert if anyone's waiting on it */
82     spinlock_cleanup(&sem->sem_lock);
83     wchan_destroy(sem->sem_wchan);
84     kfree(sem->sem_name);
85     kfree(sem);
86 }
```

`sem_destroy` fa una cleanup dello spinlock (un paio di KASSERT per assicurarsi che nessuno stia usando lo spinlock?), dealloca il wait channel del semaforo, e poi dealloca il semaforo stesso.

P (wait)

```

88 void
89 P(struct semaphore *sem)
90 {
91     KASSERT(sem != NULL);
92
93     /*
94      * May not block in an interrupt handler.
95      *
96      * For robustness, always check, even if we can actually
97      * complete the P without blocking.
98      */
99     KASSERT(curthread->t_in_interrupt == false);
100
101    /* Use the semaphore spinlock to protect the wchan as well. */
102    spinlock_acquire(&sem->sem_lock);
103    while (sem->sem_count == 0) {
104        /*
105         *
106         * Note that we don't maintain strict FIFO ordering of
107         * threads going through the semaphore; that is, we
108         * might "get" it on the first try even if other
109         * threads are waiting. Apparently according to some
110         * textbooks semaphores must for some reason have
111         * strict ordering. Too bad. :-)
112         *
113         * Exercise: how would you implement strict FIFO
114         * ordering?
115         */
116        wchan_sleep(sem->sem_wchan, &sem->sem_lock);
117    }
118    KASSERT(sem->sem_count > 0);
119    sem->sem_count--;
120    spinlock_release(&sem->sem_lock);
121 }

```

Vediamo la funzione di wait.

Essa riceve il puntatore al semaforo.

Proviamo ad acquisire uno spinlock sul semaforo stesso, per assicurarci mutua esclusione sul

counter del semaforo.

Una volta acquisito, controlliamo il counter del semaforo: finchè il counter è uguale a 0, dobbiamo aspettare.

Andiamo in stato di sleep nel wait channel del semaforo in questione, ovviamente ($\text{sem} \rightarrow \text{sem_wchan}$).

Ma se andiamo in sleep, perchè appunto dobbiamo aspettare, dovremmo anche rilasciare lo spinlock acquisito (altrimenti il thread va in sleep senza rilasciare il

lock, e gli altri thread non potrebbero acquisirlo).

Questo viene effettivamente fatto dalla wchan_sleep, che appunto vediamo che riceve come parametro anche il lock del semaforo, per rilasciarlo e provare a riacquisirlo al ritorno dallo stato di sleeping.

```
992  /*
993   * Yield the cpu to another process, and go to sleep, on the specified
994   * wait channel WC, whose associated spinlock is LK. Calling wakeup on
995   * the channel will make the thread runnable again. The spinlock must
996   * be locked. The call to thread_switch unlocks it; we relock it
997   * before returning.
998  */
999 void
1000 wchan_sleep(struct wchan *wc, struct spinlock *lk)
1001 {
1002     /* may not sleep in an interrupt handler */
1003     KASSERT(!curthread->t_in_interrupt);
1004
1005     /* must hold the spinlock */
1006     KASSERT(spinlock_do_i_hold(lk));
1007
1008     /* must not hold other spinlocks */
1009     KASSERT(curcpu->c_spinlocks == 1);
1010
1011     thread_switch(S_SLEEP, wc, lk);
1012     spinlock_acquire(lk);
1013 }
```

Più precisamente il rilascio dello spinlock viene fatto all'interno di `thread_switch`, che fa anche tante altre cose, ma per ora non scendiamo nei dettagli.

Inoltre in

`wchan_sleep` ci assicuriamo che siamo effettivamente possessori del lock, e che non possediamo altri locks (sennò vorrebbe dire che andiamo in stato di sleep mentre possediamo dei locks → deadlock per altri threads).

Ad ogni modo, una volta che il thread è in stato di sleeping, esso può essere svegliato in seguito ad una signal sul wait channel.

Può essere, però, che nel frattempo il counter del semaforo diventi di nuovo 0, e per questo abbiamo il while che va a ritestare la condizione: se è di nuovo 0, tornerò in stato di sleeping, altrimenti uscirò dal while.

Vediamo anche da qui come abbia senso che il lock sul semaforo venga riacquisito al ritorno dallo sleep (nella

`wchan_sleep`): come prima di entrare nel while, dobbiamo avere il lock per la mutua

esclusione su `sem->sem_count == 0`.

Questo verrà poi mantenuto, nel caso in cui si possa procedere nell'esecuzione (per poi essere rilasciato una volta decrementato il counter), o rilasciato nuovamente nella

`thread_switch` se dobbiamo tornare in stato di sleeping.

V (signal)

```
123 void
124 V(struct semaphore *sem)
125 {
126     KASSERT(sem != NULL);
127
128     spinlock_acquire(&sem->sem_lock);
129
130     sem->sem_count++;
131     KASSERT(sem->sem_count > 0);
132     wchan_wakeone(sem->sem_wchan, &sem->sem_lock);
133
134     spinlock_release(&sem->sem_lock);
135 }
```

Molto più semplicemente, la funzione di signal acquisisce il lock sul semaforo, incrementa il counter (assicurandosi poi che esso sia maggiore di 0) e fa una `wchan_wakeone` sul wait channel del semaforo, e rilascia il lock.

```

1015  /*
1016   * Wake up one thread sleeping on a wait channel.
1017   */
1018 void
1019 wchan_wakeone(struct wchan *wc, struct spinlock *lk)
1020 {
1021     struct thread *target;
1022
1023     KASSERT(spinlock_do_i_hold(lk));
1024
1025     /* Grab a thread from the channel */
1026     target = threadlist_remhead(&wc->wc_threads);
1027
1028     if (target == NULL) {
1029         /* Nobody was sleeping. */
1030         return;
1031     }
1032
1033     /*
1034      * Note that thread_make_runnable acquires a runqueue lock
1035      * while we're holding LK. This is ok; all spinlocks
1036      * associated with wchans must come before the runqueue locks,
1037      * as we also bridge from the wchan lock to the runqueue lock
1038      * in thread_switch.
1039      */
1040
1041     thread_make_runnable(target, false);
1042 }
```

Anche in questo caso passiamo il lock sul semaforo a `wchan_wakeone` , perchè questa è un'operazione che possiamo fare solo se siamo in possesso dello spinlock.

Locks

```

141 struct lock *
142 lock_create(const char *name)
143 {
144     struct lock *lock;
145
146     lock = kmalloc(sizeof(*lock));
147     if (lock == NULL) {
148         return NULL;
149     }
150
151     lock->lk_name = kstrdup(name);
152     if (lock->lk_name == NULL) {
153         kfree(lock);
154         return NULL;
155     }
156
157     HANGMAN_LOCKABLEINIT(&lock->lk_hangman, lock->lk_name);
158
159     // add stuff here as needed
160
161     return lock;
162 }
```

In modo simile ai semafori, lock_create alloca spazio per una struttura di tipo lock, e le assegna il nome passato come parametro.

Vediamo che il resto è da implementare..

Due possibilità: implementiamo un wait channel e un flag (0/1) come "counter" similmente a com'è stato fatto per un semaforo in

`sem_create` , oppure potremmo creare un semaforo (che appunto è già implementato) con counter a 1 (quindi un semaforo binario) e sfruttare l'implementazione dei semafori per implementare i locks!

Nella seguente trattazione consideriamo l'ultimo caso.

Vediamo quindi la soluzione del prof:

```

77  /*
78   * Simple lock for mutual exclusion.
79   *
80   * When the lock is created, no thread should be holding it. Likewise,
81   * when the lock is destroyed, no thread should be holding it.
82   *
83   * The name field is for easier debugging. A copy of the name is
84   * (should be) made internally.
85   */
86 struct lock {
87     char *lk_name;
88     // add what you need here
89     // (don't forget to mark things volatile as needed)
90 #if OPT_SYNCH
91 #if USE_SEMAPHORE_FOR_LOCK
92     struct semaphore *lk_sem;
93 #else
94     struct wchan *lk_wchan;
95 #endif
96     struct spinlock lk_lock;
97     volatile struct thread *lk_owner;
98 #endif
99 };

```

Vediamo che, se abbiamo impostato `USE_SEMAPHORE_FOR_LOCK` a 1, la definizione della struttura di un lock ora include un puntatore ad un semaforo.

Inoltre, troviamo uno spinlock e una novità: un puntatore al thread che possiede il lock.

Infatti, il lock introduce il concetto di possesso: solo il thread che possiede il lock può fare la signal, mentre nei semafori chiunque poteva fare signal (qualunque dei threads entrati in sezione critica nel caso di semaforo non binario).

Il puntatore al thread owner del lock è dichiarato volatile per garantire che sia condiviso, quindi sia presente in memoria e non in un registro della CPU.

lock_create

```

141 struct lock *
142 lock_create(const char *name)
143 {
144     struct lock *lock;
145
146     lock = kmalloc(sizeof(*lock));
147     if (lock == NULL) {
148         return NULL;
149     }
150
151     lock->lk_name = kstrdup(name);
152     if (lock->lk_name == NULL) {
153         kfree(lock);
154         return NULL;
155     }
156
157     // add stuff here as needed
158
159 #if OPT_SYNCH
160 #if USE_SEMAPHORE_FOR_LOCK
161     lock->lk_sem = sem_create(lock->lk_name,1);
162     if (lock->lk_sem == NULL) {
163 #else
164     lock->lk_wchan = wchan_create(lock->lk_name);
165     if (lock->lk_wchan == NULL) {
166 #endif
167         kfree(lock->lk_name);
168         kfree(lock);
169         return NULL;
170     }
171     lock->lk_owner = NULL;
172     spinlock_init(&lock->lk_lock);
173 #endif
174     return lock;
175 }
```

Nella creazione del lock notiamo la creazione di un semaforo binario, con il nome del lock stesso, e poi l'inizializzazione dell'owner del lock (NULL, visto che ancora nessuno lo possiede) e dello spinlock.

lock_acquire

```

195 void
196 lock_acquire(struct lock *lock)
197 {
198     // Write this
199 #if OPT_SYNCH
200     KASSERT(lock != NULL);
201     if (lock_do_i_hold(lock)) {
202         kprintf("AAACKK!\n");
203     }
204     KASSERT(!lock_do_i_hold(lock));
205
206     KASSERT(curthread->t_in_interrupt == false);
207
208 #if USE_SEMAPHORE_FOR_LOCK
209 /*
210  * G.Cabodi - 2019: P BEFORE!!!! spinlock acquire. OS161 forbids sleeping/releasing
211  * the CPU while owning a spinlocks: this could be a cause of deadlock.
212  * THE spinlock passed to wchan_wait is the only one allowed.
213  * This is checked in various parts of the code (see for instance wchan_sleep.
214  * as P may result in "wait", it cannot be called while owning the spinlock.
215 */
216     P(lock->lk_sem);
217     spinlock_acquire(&lock->lk_lock);
218 #else
219     spinlock_acquire(&lock->lk_lock);
220     while (lock->lk_owner != NULL) {
221         wchan_sleep(lock->lk_wchan, &lock->lk_lock);
222     }
223 #endif
224     KASSERT(lock->lk_owner == NULL);
225     lock->lk_owner=curthread;
226     spinlock_release(&lock->lk_lock);
227 #endif
228     (void)lock; // suppress warning until code gets written
229 }
```

Come possiamo acquisire il lock?

Sfruttando la funzione `P` del semaforo!

Non è altro che la funzione che abbiamo visto prima: manderà il thread in sleep finchè il lock non è disponibile, se lo è invece lo rende non disponibile per tutti gli altri e consente di proseguire nell'esecuzione.

In particolare, dopo aver acquisito il lock, impostiamo subito il thread corrente come owner del lock.

Lo

`spinlock_acquire(&lock->lk_lock)` , anche se è appurato che siamo proprietari del lock e in teoria gli altri threads non dovrebbero star facendo nulla con il lock, va messo per sicurezza, dato che comunque stiamo modificando l'owner del lock.

lock_release

```
231 void
232 lock_release(struct lock *lock)
233 {
234     // Write this
235 #if OPT_SYNCH
236     KASSERT(lock != NULL);
237     KASSERT(lock_do_i_hold(lock));
238     spinlock_acquire(&lock->lk_lock);
239     lock->lk_owner=NULL;
240     /* G.Cabodi - 2019: no problem here owning a spinlock, as V/wchan_wakeone
241      do not lead to wait state */
242 #if USE_SEMAPHORE_FOR_LOCK
243     V(lock->lk_sem);
244 #else
245     wchan_wakeone(lock->lk_wchan, &lock->lk_lock);
246 #endif
247     spinlock_release(&lock->lk_lock);
248 #endif
249     (void)lock; // suppress warning until code gets written
250 }
```

Nella funzione `lock_release` semplicemente impostiamo a NULL l'owner del lock e chiamiamo la funzione `V` (signal).

Impostiamo l'owner a NULL perchè questa condizione viene usata dai threads per capire se il lock è disponibile o meno ($lk_owner \neq \text{NULL} \rightarrow \text{lock non disponibile} \rightarrow \text{wait}$).

```

253 bool
254 lock_do_i_hold(struct lock *lock)
255 {
256     // Write this
257 #if OPT_SYNCH
258     bool res;
259     /* G.Cabodi - 2019: this could possibly work without spinlock for mutual
260      exclusion, which could simplify the semaphore-based solution, by
261      removing the spinlock.
262      Whenever the current thread owns the lock, the test is safe without
263      guaranteeing mutual exclusion.
264      If NOT the owner, a wrong verdict could happen (very low chance!!!)
265      by wrongly reading a pointer == curthread. However, using the spinlock
266      is good practice for shared data. */
267     spinlock_acquire(&lock->lk_lock);
268     res = lock->lk_owner == curthread;
269     spinlock_release(&lock->lk_lock);
270     return res;
271 #endif
272     (void)lock; // suppress warning until code gets written
273     return true; // dummy until code gets written
274 }

```

Notiamo inoltre che la funzione `lock_do_i_hold` è stata modificata adhoc per il lock: abbiamo detto che siamo possessori del lock se il thread corrente è effettivamente quello memorizzato come owner nella struttura del lock.
Quindi è stato introdotto questo controllo.

Condition variable

Example: wait on condition

```
use_a_thing {
    lock_acquire(mylock); /* lock out others */
    if(x == 0 && (y > 0 || z > 0))
        P(no_go);
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}
```

Con le **condition variable** aspettiamo l'avverarsi di una condizione.

Tale condizione si basa su alcune variabili.

Nell'esempio, se

x == 0 && (y>0 || z>0) allora wait.

Example: wait on condition



```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* semaphore to wait if necessary */
struct semaphore *no_go = sem_create("MySem", 0);

compute_a_thing {
    lock_acquire(mylock); /* lock out others */
    /* compute new x, y, z */
    x = f1(x); y = f2(y); z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0)) V(no_go);
    lock_release(mylock); /* enable others */
}
```

D'altro canto un altro thread starà lavorando su tali variabili, e, se alla fine del calcolo, la condizione di wait è soddisfatta, allora si fa una signal.

Abbiamo visto come le primitive di wchan_sleep, wchan_wakeone e wchan_wakeall siano essenzialmente le primitive che implementano le funzionalità di wait e signal.

Tali funzioni prendono come parametro anche uno spinlock, che appunto viene rilasciato quando il thread va in sleep e riacquisito al suo ritorno.

Le condition variable hanno delle primitive assolutamente analoghe, ma che ricevono come parametro un lock piuttosto che uno spinlock:

```

326 void
327 cv_wait(struct cv *cv, struct lock *lock)
328 {
329     // Write this
330 #if OPT_SYNCH
331     KASSERT(lock != NULL);
332     KASSERT(cv != NULL);
333     KASSERT(lock_do_i_hold(lock));
334
335     spinlock_acquire(&cv->cv_lock);
336     /* G.Cabodi - 2019: spinlock already owned as atomic lock_release+wchan_sleep
337      needed */
338     lock_release(lock);
339     wchan_sleep(cv->cv_wchan,&cv->cv_lock);
340     spinlock_release(&cv->cv_lock);
341     /* G.Cabodi - 2019: spinlock already released to avoid ownership while
342      (possibly) going to wait state in lock_acquire.
343      Atomicity wakeup+lock_acquire not guaranteed (but not necessary!) */
344     lock_acquire(lock);
345 #endif
346
347     (void)cv;    // suppress warning until code gets written
348     (void)lock; // suppress warning until code gets written
349 }

```

Alla fine l'unica differenza tra wait channel e condition variable è che: nel caso di wait channel la condizione che deve essere osservata viene osservata tramite spinlocks, e nel caso di condition variable tramite locks.

NOTA: I semafori sono realizzati in modalità “NON busy waiting”. Per realizzarli si ricorre a “wait_channel”, che sono un tipo di condition variable realizzato in OS161, ASSOCIATO a spinlock. Siccome gli spinlock sono lock con “busy waiting”, essi sono adatti a casi in cui (all’interno del kernel) si preveda attesa limitata. Si legga (e si tracci con il debugger) la realizzazione di un semaforo: si potrà osservare che lo spinlock protegge (mediante mutua esclusione) l’accesso al contatore interno al semaforo, mentre il wait_channel serve per effettuare attese e segnalazioni (cioè un thread aspetta di essere svegliato, mediante segnalazione, da un altro thread).

ATTENZIONE: un lock non è unicamente un semaforo binario. A differenza del semaforo, il lock ha il concetto di “possesso” (*ownership*), cioè `lock_release()` può essere fatto unicamente dal thread che ha ottenuto il lock mediante `lock_acquire()` e attualmente ne è owner (Errore frequente: considerare come owner il thread che chiama `lock_create()`). Questo implica che la struct `lock` deve avere un puntatore al thread owner: si ricorda che è disponibile il simbolo `curthread` (non è una variabile globale ma una #define fatta in `current.h`, utilizzabile come se si trattasse di una variabile globale). Il tentativo di `lock_release()` da parte di un thread non owner può (a scelta) essere gestito come un errore fatale (KASSERT oppure panic) oppure essere semplicemente ignorato, non rilasciando il lock. Si consiglia KASSERT in quanto, trattandosi di thread kernel, un tentativo di `lock_release()` errato è un vero e proprio errore nel kernel.

Si noti inoltre che occorre realizzare la funzione `lock_do_i_hold()`, che deve indicare al programma chiamante se il thread corrente sia o meno owner di un lock ricevuto come parametro. Si tratta con tutta probabilità di leggere un puntatore memorizzato nella struct `lock`: la lettura andrebbe quindi questo puntatore andrebbe gestita in mutua esclusione (o quanto meno occorre considerare se l'accesso multiplo da più thread sia o meno critico). Attenzione, qualora si utilizzi uno spinlock per realizzare la `lock_do_i_hold` come sezione critica, al fatto che OS161 non consente di attendere su uno spinlock (chiamare `spinlock_acquire()`) se il thread corrente sia già owner di un altro spinlock.

Per completezza si descrivono in seguito le *condition variable*. Per una trattazione più completa, si può far riferimento a [https://en.wikipedia.org/wiki/Monitor_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization)) ed alla lezione “os161-synchro”.

Una *condition variable* è essenzialmente una primitiva di sincronizzazione che consente di attendere che una condizione (eventualmente falsa al momento attuale) diventi vera.

Una condition variable è sempre accompagnata da un lock (**ATTENZIONE, UN LOCK, mentre al wait channel è associato uno spinlock**), passato come parametro alle funzioni, che ne garantisce l'accesso protetto in mutua esclusione).

Le funzioni fornite dalle *condition variable* in OS161 sono:

- `cv_wait()`: rilascia il *lock* ricevuto come parametro (che deve essere stato acquisito in precedenza dal thread chiamante), e si mette in attesa di una `cv_signal()` o `cv_broadcast()`.
- `cv_signal()` e `cv_broadcast()` svolgono lo stesso compito, ma differiscono nel numero di thread svegliati, la prima ne sveglia solo uno (tra quelli in attesa) la seconda tutti.

Una *condition variable* potrebbe essere realizzata ricorrendo a *semafori*: l'attesa (`cv_wait()`) potrebbe essere realizzata mediante una `P()`, mentre `cv_signal()` mediante `V()` sul *semaforo*. Più complicata sarebbe la realizzazione di `cv_broadcast()`, che richiederebbe un contatore delle *wait* in corso. Inoltre, esisterebbe un problema di semantica: coi *semafori*, si dovrebbe gestire il fatto che una `V()` verrebbe eventualmente "ricordata" (nel caso di nessun thread in attesa) e potrebbe sbloccare una futura `P()`: tale comportamento non è corretto con le *condition variable* (`cv_signal()` e `cv_broadcast()` sbloccano unicamente thread in attesa ora, non in un momento futuro). DETTO IN ALTRI TERMINI: la *condition variable* realizzata con i semafori si comporterebbe in modo leggermente diverso da COME DOVREBBE.

Si consiglia pertanto di realizzare le *condition variable* direttamente mediante *wait_channel* (e *spinlock*). Siccome il *wait channel* ha una semantica sostanzialmente simile a quella delle *condition variable* (ma ha uno *spinlock* anziché un *lock*), si può utilizzare un *wait_channel* per le segnalazioni, ma occorre gestire (in modo corretto) un *lock*. I programmi di test per *condition variable* sono `sy3` e `sy4`, che chiamano le funzioni `cvttest()` e `cvttest2()`.

I file contenenti definizioni e funzioni sulle *condition variable* sono `kern/include/synch.h` e `kern/thread/synch.c`. Il file contenente `cvttest()` e `cvttest2()` è `kern/test/synctest.c`.

*ATTENZIONE: sia il rilascio del lock, sia la messa in attesa del thread (nella `cv_wait()`), vanno fatti in modo atomico, evitando cioè che un altro thread acquisisca il lock prima che il thread vada in attesa (sul *wait_channel*). Si consiglia, a tale scopo, di sfruttare lo *spinlock* associato al *wait_channel*.*

OLD

OS/161 Spinlocks (kern/thread/spinlock.c)

```
spinlock_acquire(struct spinlock *splk) {
    ...
    while (1) {
        /* Do test-test-and-set, that is, read first before doing test-and-set, to
         * reduce bus contention.
         * Test-and-set is a machine-level atomic operation
        */
        if (spinlock_data_get(&splk->splk_lock) != 0) {
            continue;
        }
        if (spinlock_data_testandset(&splk->splk_lock) != 0) {
            continue;
        }
        break;
    }
    ...
}
```

OS161 ha già 2 primitive di sincronizzazione implementate: i semafori e gli spinlocks.

Ne esistono altre e l'obiettivo del lab è implementare queste altre.

Lo Spinlock è una primitiva che consente la mutua esclusione: il processo che richiede accesso alla sua sezione critica, se nessuno sta usando la sezione critica, può accedervi.

Quando arriva un secondo processo che vuole accedervi, questo deve chiedere se è disponibile lo spinlock, cioè questa variabile che viene data al thread che può accedere alla sezione critica e ha delle caratteristiche importanti:

- Lo spinlock può essere rilasciato solo dal thread che l'ha acquisito
- Gli altri thread che non possono acquisire lo spinlock looppano continuamente in una condizione di test-and-set finchè lo spinlock non sarà nuovamente disponibile: cioè il thread continua a fare questo loop inattivo occupando cicli di cpu finchè lo spinlock non sarà libero.

Questa soluzione di continuare a far attendere il thread, occupando anche cicli cpu, non è ottimale per le grandi sezioni critiche perchè se mi aspetto che la sezione critica sia disponibile a breve allora okay, ci sta che non metto in pausa il thread ed evito tutto l'overhead che ne deriverebbe (tanto dopo poco prenderà lo spinlock), ma se la sezione critica richiede tanto tempo per essere eseguita e quindi non ho un cambio di spinlock rapido, continuare ad occupare cicli di cpu inutilmente — non è ottimale.

Infatti per le sezioni critiche più grandi si utilizza il Lock, un'altra variabile che

vedremo dopo e che dovremo implementare, che mette in sleep il thread che non può ancora accedere alla sezione critica.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- A semaphore **s** is an integer variable that, apart from initialization is accessed through two standard **atomic** operations:
 - **wait()** and **signal()**
- Definition of the **wait()**

```
wait(s) {  
    while (s <= 0);  
    // busy wait  
    s--;  
}
```
- Definition of the **signal()**

```
signal(s) {  
    s++;  
}
```

I semafori proteggono anch'essi l'accesso alla sezione critica, ma in modo diverso: può esserci mutua esclusione ma non per forza (in lock e spinlock solo un thread può entrare in sezione critica), mentre i semafori hanno una variabile interna detta counter, che in generale quando è maggiore di 0, i threads possono entrare in sezione critica e quando entrano decrementano il counter, finché questo non sarà 0 e quindi i threads non potranno entrare.

Il counter viene incrementato quando un thread esce dalla sezione critica.

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Behave similarly to **mutex lock**
- Can solve various synchronization problems
- Consider **P₁** and **P₂** that require **S₁** to happen before **S₂**
Create a semaphore "synch" initialized to 0
 - P1:

```
s1;  
signal(synch);
```
 - P2:

```
wait(synch); s2;
```
- Can implement a counting semaphore **S** as a binary semaphore

Il counter dei semafori è inizializzato alla creazione del semaforo, e in base al valore iniziale del counter possiamo eseguire mutua esclusione (es. di semaforo binario: se ho un counter inizializzato a 1, ciò vuol dire che solo un thread alla volta può entrare).

Qual è allora la differenza principale tra un semaforo binario e un lock o spinlock?
Gli ultimi due hanno il concetto di proprietà: il lock può essere rilasciato solo dal thread che l'ha acquisito, mentre il semaforo non ha questa limitazione.

Condition Variables

- While mutexes implement synchronization by controlling thread access to data, **condition variables allow threads to synchronize based upon the actual value of data.**
- In a critical section, a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.
 - It will suspend by waiting on a condition variable
 - It will release the critical section lock (MUTEX)
 - When that condition variable is signaled, it will become ready again. It will attempt to reacquire that critical section lock and only then it will be able to proceed

L'ultima variabile di sincronizzazione che vediamo oggi (anch'essa da implementare, insieme ai Locks) sono le **Condition Variables**.

Sono anch'esse primitive di sincronizzazione per eseguire l'accesso alle sezioni critiche con un grado di libertà in più perchè consente l'accesso alla sezione critica in base ad una condizione: se la condizione è soddisfatta (if x == true) allora entro nella sezione.

A differenza degli altri metodi dove non è possibile eseguire un check su una condizione.