**­­PdS** **2023 - Laboratorio di OS161 – 3**

***Per affrontare questo laboratorio è necessario:***

* ***aver svolto (e capito) il laboratorio 1. Il laboratorio 2, pur se importante, non ha implicazioni dirette su questo laboratorio, che non ha bisogno di processi utente e system call.***
* ***aver visto le lezioni os161-synchro, avendo chiara la semantica di semafori, lock e condition variable, nonché il funzionamento di spinlock e di wait channels in OS161.***
* ***usare il debugger.***

***Nota sull’uso delle opzioni:*** *come detto al termine della demo “os161-synchro-demo”, si consideri il fatto che semafori, e soprattutto lock, sono già utilizzati nel kernel OS161, in particolare nel boot, come pure (semafori) nelle istruzioni di I/O sulla console (i lock non funzionano correttamente, ma questo non pregiudica il funzionamento del kernel, che non è particolarmente critico, nella versione base, in relazione a multi-threading). Se tuttavia si modificasse l’implementazione dei semafori, e/o si realizzassero i lock, si consideri che il kernel potrebbe “piantarsi” immediatamente senza completare il boot. Per questo è* ***importante poter facilmente isolare*** *le varie parti di codice aggiunto, eventualmente tramite* ***opzioni e compilazione condizionale****. Si consiglia inoltre VIVAMENTE di procedere passo-passo, se possibile, con implementazioni FACILI/SEMPLICI in prima battuta, passando a quelle più critiche solo una volta messe a punto e verificate (eventualmente con debug) le prime. Si tenga poi presente che, in un’esecuzione con GDB, la combinazione di tasti Ctrl-G (control-G) nella finestra di esecuzione (la console) permette di interrompere l’esecuzione, passando il controllo al debugger: può tornare utile in alcuni casi (non tutti, dipende dall’entità del problema) in cui un programma sia in loop infinito o altri errori.*

# **Implementazione di lock e condition variable (kernel-side)**

# 

Si faccia riferimento ai precedenti laboratori di OS161, nei quali si è richiesto di effettuare test di esecuzione di thread kernel. NON si affronta quindi il problema di processi e/o thread user e della loro sincronizzazione.

**Test di sincronizzazione ed implementazione dei lock**

Provare i test sy1 e sy2 e tracciarne, utilizzando il debug, l’esecuzione. sy1 effettua un test di sincronizzazione mediante *semafori*, sy2 mediante *lock*. I *semafori* sono già realizzati in OS161, mentre i *lock* NO.

***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).*

Si chiede di realizzare i *lock* in due modi diversi (usare le opzioni di *conf.kern* e la compilazione condizionale per differenziare le due realizzazioni):

1.      utilizzando *semafori*: di fatto un *lock* è un *semaforo binario* (contatore con valore massimo 1). Si tratta di modificare la struct lock in kern/include/synch.h, e completare le funzioni lock\_create(), lock\_destroy(), lock\_acquire(), lock\_release(), lock\_do\_i\_hold() in kern/thread/synch.c.

2.      ricorrendo direttamente a *wchan* e *spinlock*. Questa è sicuramente la versione preferibile, per la quale si consiglia di far riferimento, adattandola opportunamente, alla realizzazione INTERNA dei *semafori* (si tratta, QUASI, di copiarla, FACENDO ATTENZIONE ALLE DIFFERENZE).

***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.*

**Test di sincronizzazione e implementazione delle condition variable**

Anche le *condition variable*, come i *lock*, vanno realizzate, in quanto in OS161 base, le funzioni di sincronizzazione con condition variable sono “vuote”. Provare i test sy3 e sy4 e tracciarne, utilizzando il debug, l’esecuzione. I test dovrebbero segnalare eventuali problemi nella realizzazione di *condition variable*, tuttavia sono ancora possibili errori non segnalati da tali test (in particolare sy3, programma di test più semplice).

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)> 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 cvtest() e cvtest2().

I file contenenti definizioni e funzioni sulle *condition variable* sono kern/include/synch.h e kern/thread/synch.c. Il file contenente cvtest() e cvtest2() è kern/test/synchtest.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.*