**Come viene fatto lo yield()?**

Bisogna salvare il contesto del thread corrente per poi ripristinare il contesto dello scheduler

Lo yield viene eseguito tramite sigsetjmp che permette di salvare il contesto. Il tutto deve essere eseguito in modo atomico disabilitando gli interrupt e riabilitandoli alla fine. Non abbiamo il problema del cuscinetto siccome salviamo il contesto nello heap

**Come faccio a capire se il thread si è appena svegliato e quindi deve eseguirlo oppure se ha appena eseguito e deve passare la palla allo scheduler?**

Guardo il valore di ritorno di **sigsetjmp**. Se 0 vuol dire che il thread ha appena eseguito e deve passare la palla, altrimenti vuol dire che è appena stato schedulato per essere eseguito.

**Cosa significa cooperativo?**

Ogni thread mantiene uno stack contest ma è necessario passare esplicitamente ad altri thread.

**Come viene fatto il create di un thread?**

Creo un nuovo thread inizializzando tutti i suoi parametri di default (stato ready=0), accetta come parametro gli attributi di quest’ultimo (ma viene ignorato perché si vuole tenere compatibilità con pthread), un puntatore alla funzione start routine ed un altro gli argomenti della start. Dopo aver settato questi valori di default aggiunge il thread nella coda dello scheduler. NB che viene solo creato non instanziato

**Cosa serve bthread\_get\_scheduler()?**

Serve ad inizializzare una sola volta lo sheduler e a ritornare sempre lo stesso elemento alle chiamate succesive.

**Cosa serve set\_priority(int a)?**

Serve ad impostare la proprietà di priorità del thread corrente (quindi bisogna farlo nella funzione di start routine)

**Cosa fa bthread\_sleep(double ms)?**

Mette a dormire il thread chiamante impostando nelle proprietà lo stato di sleeping ed impostare la proprietà di wake\_up contenente il tempo corrente e i ms, successivamente deve fare uno sleep altrimenti non cambia

**Cosa fa cancel()?**

Cancel prende il thread dalla coda e va a settare il flag di cancellazione.

**Cosa fa testCancel()?**

Va a controllare se deve suicidarsi, perchè se ha la flag di cancellazione settata termina tramite il metodo exit.

**Cosa fa exit()?**

Imposta lo stato del thread chiamante a zombie (nella join, verrà fatto un controllo se zombie e in caso affermativo il thread viene fisicamente cancellato). Inoltre salva il valore di ritorno in retval. Alla fine dovrò fare un yield.

**Cosa fa bthread\_printf()?**

La funzione printf non è thread-safe, ne definiamo una apposita che lavora in modo atomico (bloccano il timer) e che permette ai thread di stampare su stdout.

**Cosa fa bthread\_setup\_timer()?**

In caso non sia ancora stato inizializzato il timer lo inizializza in modo che venga inviato un segnale periodico che si ripete ogni quantum di tempo. Viene assegnato come handler del timer (tramite segnale) la funzione di yield. Cosi facendo abbiamo implementato la preemption.

**Cosa fa block\_timer\_signal?**

Crea una maschera, in caso non sia già stata inizializzata (mask = null), che blocca i segnali del timer (sig\_block). Idealmente serve a disabilitare la preemption in modo che le righe di codice a seguire dalla chiamata di questa funzione eseguano in modo atomico.

**Cosa fa unblock\_timer\_signal?**

Imposta la maschera di default, quindi abilita i segnali del timer e in pratica riabilito la preemption.

**Cosa fa la join?**

**La funzione principale della join è far partire i thread e controllare se sono nello stato di zombie. Cosa vuol dire che, se sono nello stato di zombie ritorni 0 (e quel thread ha finito) altrimenti fa la schedulazione vera e propria. Chiama la politica di scheduling ma solo dopo aver salvato il contesto dello scheduler in modo che poi fa la schedulazione dei thread e fa partire i thread e poi quando li fa partire fanno lo yield**

Nell’attesa che termini il thread specificato come parametro (che raggiunga lo stato zombie) schedula e fa eseguire tutti i thread nella coda. Per prima cosa vengono disabilitati gli interrupt timer per eseguire in modo atomico. Controlla che il thread passato come argomento non abbia finito l’esecuzione

**Cosa fa la lottery\_scheduling?**

Sommo tutte le priorità di tutti i thread a ready (in modo da avere un numero massimo di biglietti) e assegno tramite un array il possedimento del singolo biglietto al thread e controllo se il thread dorme forse è il momento di svegliarsi, alla prossima estrazione della lotteria rientrerà nell'array. Si effettua l’estrazione tramite un numero random e chi esce sarà il thread utilizzato.

**Cosa fa la random\_scheduling?**

Prende thread a caso dalla coda (offset) finchè non ne trova uno pronto ad eseguire, ogni volta che ne estrae uno controlla che non sia il momento di svegliarsi se sta dormendo.

**Cosa fa la priority\_scheduling?**

Fa eseguire un thread per tanti quanti di tempo quanto è la sua priorità, quando un thread non è più ready oppure termina i quanti di tempo, resetta i quanti e schedula il successivo con round robin., altrimenti incrementa il quantum.

**Cosa fa la round\_robin\_scheduling?**

Semplicemente itera la coda ricercando il primo thread con lo stato ready e lo assegna come current item. Inoltre controlla anche se ci sono thread che devono svegliarsi, in caso affermativo li sveglia (tramite controllo dello stato e il tempo di risveglio < tempo corrente)

**Cosa fa set\_schedulong\_routine?**

Setta la politica di scheduling dello scheduler, accetta infatti un puntatore a funzione la quale verrà invocata ogni volta che lo scheduler dovrà scegliere quale thread eseguirà

**Cosa fa get\_queue\_at?**

Ritorna una vista della coda che parte dal thread specificato come parametro della funzione. Facendo un for per tutta la coda e spostando tramite offset il thread corrente. Se la coda non contiene questo thread ritorna NULL.

**Cosa fa check\_if\_zombie?**

Controlla se il thread passato come argomento è nello stato zombie, in caso negativo la funzione ritorna 0. Altrimenti, in caso in la variabile retval non sia NULL (quindi ci aspettiamo un valore di ritorno) il valore di ritorno del thread viene salvato. Successivamente libero la memoria (free dell’elemento) e tolgo il thread dalla coda. Infine ritorno 1.

Barriera

Init 🡪 Inizializza una barriera controllando che non sia NULL, poi imposta la dimensione della barriera a count,quando il numero di thread in coda raggiunge questo valore li sblocca tutti.

bthread\_barrier\_destroy🡪Controlla che non ci sia nessuno ad aspettare alla barirera, poi libera la memoria occupata da quest'ultima.

bthread\_barrier\_wait🡪 Per evitare context switch vengono disabilitati i segnali del timer, così facendo ci assicuriamo che le operazioni a seguire vengano svlote in modo atomico. Successivamente controlliamo che il thread chiamante non sia l'ultimo che stiamo aspettando, in caso facciamo ripartire tutti e resettiamo la barriera. Altrimenti blocchiamo il thread chiamante e lo mettiamo in lista incrementando il numero di thread in coda alla barriera. Infine il thread corrente fa yeld per mettersi a "dormire"

Condition

Init

Inizializza la condition con una coda vuota assicurandosi che sia correttamente instanziata.

Destroy

Distrugge la condition controllando che non ci sia nessuno in attesa di esssere svegliato

Wait

Per evitare context switch vengono disabilitati i segnali del timer, così facendo ci assicuriamo che le operazioni a seguire vengano svlote in modo atomico. Assumendo che per aspettare su una variabile di condizione occorra essere in possesso di un mutex la prima cosa che andiamo a fare è controllare che il thread chiamante sia effettivamente il proprietario del mutex passato.In caso affermativo il mutex viene rilasciato (affinche altri possano acquisirlo ed eventualmente mettersi a dorimire sulla condition). Infine il thread viene bloccato e messo in coda di attesa, infatti vi è uno yield. Quando il thread viene svegliato la prima cosa che fa sarà riacquisire il mutex che aveva rilascliato prima di mettersi a dormire sulla condizione.

Signal

Per evitare context switch vengono disabilitati i segnali del timer, così facendo ci assicuriamo che le operazioni a seguire vengano svlote in modo atomico. Viene recuperato il primo elemento della coda di attesa e viene svegliato. Poi il thread chiamante fa un yield per generosità.

Broadcast

Per evitare context switch vengono disabilitati i segnali del timer, così facendo ci assicuriamo che le operazioni a seguire vengano svlote in modo atomico. Tutti i thread nella waiting list vengono liberati, successivamente il thread chiamante fa uno yield per fare eseguire i thread appena risvegliati.

MUTEX

**Lock** Per evitare context switch vengono disabilitati i segnali del timer, così facendo ci assicuriamo che le operazioni a seguire vengano svlote in modo atomico. Viene recuperato il thread chiamante tramite lo scheduler e se il mutex è libero il thread ne diventa il proprietario. Poi vengono riabilitati gli interrupt e il thread corrente continua la sua esecuzione. Se invece il mutex è bloccato il thread corrente verrà messo nella coda di attesa e non gli sarà permesso di continuare l'esecuzione, infatti come ultima operazione si fa uno yeld.

**Trylock**

Molto simile alla funzione precedente, con la differenza che se il mutex è già bloccato il thread non viene sospeso ma semplicemente viene ritornato un valore di -1 che indica che il thread chiamante non è riuscito ad acquisire il mutex.

**Unlock**

Sblocca il mutex m. Come sempre, per eseguire in modo atomico, vengono disabilitati gli interrupt del tiemr.

Successivamente, se il thread chiamante è l'attuale proprietario del mutex quest'ultimo viene rilasciato.

In caso ci sia uno o più thread in attesa di acquisire il mutex questo viene assegnato al primo thread che lo

aveva richiesto, ovvero quello in testa alla coda di attesa.