Josep Maria Olivé i Albert Vilardell

Document Final:

El nivell que hem assolit és el 3.

Pel que fa al document de disseny, hem fet força modificacions doncs ens hem trobat amb nous problemes, a continuació les tenim totes classificades segons el grup al que pertanyen:

1. TCB → Hem eliminat totalment les estadístiques, ja que aquestes al final no ens van fer cap servei al calcular les càrregues de treball d'una altre manera. També vam decidir eliminar el struct storage, i guardar els valors dels registres en la pila de sistema del thread. Pel que fa al errno i el kernel_esp aquests si que els guardem al TCB, però directament com a atributs. La variable userStack ara es diu pag_userStack, i conté el número de pàgina lògica en la que es troba la pila d'usuari. Això es deu a que és necessari saber quina pàgina s'utilitza per la pila d'usuari, per altres crides a sistema (per exemple, pthread_exit). A més a més, hem agregat els següents nous atributs al TCB, que no hi eren anteriorment:
    a. struct task_struct *Dad → Punter al PCB del procés pare del thread.
    b. int joinable → Indica si es pot fer join amb un procés o no.
    c. struct list_head notifyAtExit → llista en la que es guarden els threads que té bloquejat el thread del TCB en qüestió. Quan aquest thread acabi la seva execució, es despertaran tots els threads que contingui la llista.
2. PCB → En aquest cas només hem eliminat les estadístiques.
3. enum state_t → Hem afegit el estat ST_ZOMBIE, un estat en el que el thread no s'executa, però el TCB tampoc està lliure, doncs aquest conté el resultat de la seva execució i s'espera a que un altre thread la reculli.
4. Semàfors → Hem afegit l'atribut in_use, que indica si un semàfor està en ús no. Aquesta és una mitigació per evitar que un usuari manipules una variable sem_t id i uses un semàfor no inicialitzat sense voler.
5. protected_threads → Hem canviat la mida del vector, ara ja no té NR_THREADS+2 elements, sinó només NR_THREADS elements. La mitigació per protegir algunes regions de memòria feien que el nostre ZeOS falles, i per tant vam haver d'eliminar-les. A més a més, l'atribut que ara està definit a la secció '.data.task' és aquest, en canvi de protected_tasks. D'aquesta manera, ens assegurem que la pila de sistema de cada thread està aliniada a 4 KB.
6. Scheduling → Com ja havíem comentat, hem agregat el nou estat zombi pels threads. En crear un thread amb pthreds_create() i finalitzar l'execució del mateix, ja sigui amb pthreads_exit() o mitjançant un retorn normal (que nosaltres hem anomenat pthreads_ret()) es guarda el valor de retorn al TCB. El thread passa a un estat de zombi, i per tant el TCB no queda lliure. Per tal

d'alliberar el TCB i obtenir el resultat del thread, algun altre thread del procés, típicament el pare, haurà d'executar un pthreads_join(). En el cas que es fes un exit, s'alliberaven de cop tots els TCB de tots els threads del procés, i per tant es finalitzaria l'execució d'un procés.

7. Thread return → Quan un thread finalitza la seva execució mitjançant un return, i no un pthreads_exit(), també hem de guardar el resultat de retorn de la funció al TCB. Per tal de fer això, hem creat una aproximació comuna. Primer de tot, en el pthread_exit() es guarda al atribut result del TCB el que es passa com a paràmetre d'aquesta crida. Alternativament, es crida la funció pthreads_ret() com a direcció de retorn de la funció d'usuari, aquesta crida a sistema en el seu wrapper fa un push de %eax, que conté el resultat de la funció executada. Posteriorment a pthreads_ret(), accedim a la pila d'usuari i busquem la direcció en la que sabem que s'ha guardat %eax per tal de moure-la al atribut result. Per tal d'aconseguir que s'executi pthreads_ret() des de mode usuari, passem com a paràmetre ocult (no modificable per l'usuari) un punter cap a aquesta funció en el pthreads_create(), i modifiquem la pila d'usuari del thread per deixar-hi el punter. La implementació recau en el wrapper de la crida a sistema: s'encarrega d'aconseguir i d'enviar en el registre %esi la direcció de memòria de pthreads_ret().
   En segon lloc, tant pthreads_exit() com pthreads_ret() criden a la funció zombify_and_wakeup(). Aquesta funció el que fa és moure el TCB a un estat de zombi, i despertar tots els threads que estan a la llista notifyAtExit.
8. Càrregues de treball → En canvid'utilitzar la crida gettime(), vem utilitzar la nova crida getticks(). La vem programar nosaltres. Retorna el nombre de ticks i utilitza unsigned long. També vem programar ltoa(), és a dir, la versió long de itoa(). El motiu del canvi està en que ens ha acabat interessant més utilitzar una crida nova, que funciona amb ticks.

A continuació tenim la taula amb les tasques que havíem que dur a terme:
Llegenda: verd tasca realitzada, taronja tasca en procés, vermell tasca pendent

Tasca	**Implementació**
Creació estructures threads	
Pthread_create i Pthread_exit	
Planificador multi-threading	
Rutines de servei multi-threading	
Testing 1: Validació canvis al codi original pel multi-threading	
Pthread_join (Nivell 1 acabat)	

Testing 2: funcions Pthread	
Codi de semàfors (nivell 2)	
Testing 3: semàfors	
Càrregues de treball	

Voldríem destacar el fet que vem fer més tasques de les que ens havíem plantejat en un principi. Especialment, en quant al Testing. Hem acabat fent 12 jocs de prova diferents. Vem decidir evaluar a fons el projecte i intentar trobar els màxims errors. Per a més informació, al ftixer user.c es poden els jocs de prova, amb una breu descripció al principi de cada funció. Per a poder escollir el joc de proves, només fa falta modificar la variable selected, localitzada en el main, i donar-li com a valor el joc de proves desitjat.

Per ser més precisos, i tal i com vem fer en l'entrega de seguiment, afegim una taula on avaluem l'estat dels fragments del codi més importants:

Fragment de codi	**Implementació**	**Jocs de proves**
pthreads_create()		
pthreads_join()		
pthreads_exit()		
sys_fork()		
sys_exit()		
sem_init()		
sem_wait()		
sem_post()		
sem_destroy()		
scheduling()		