venerdì 18 gennaio 2019

Appunti e definizioni SO

Stack and Content Switch

Processo: è un programma in esecuzione.

Contesto di un processo: Insieme di registri CPU e memoria dello stesso (es. Stack, .text, .data, .bss)

Come simulare il Contest Switch con la libreria UContext di C: si usa una struct ucontext_t per salvare il contesto, alcuni campi devono essere settati manualmente. *.uc_stack.ss_sp è il campo che indica la Stack privata del contesto, *.uc_stack.ss_size è il campo che indica la size della stack, *.uc_stack.ss_flags flags varie e *.uc_link è il riferimento al contesto da cui riprendere la computazione nel momento in cui il contesto in questione termina la sua computazione.

Funzioni utili libreria UContext: getContext(ucontext_t* ucp) salva il contesto in esecuzione nella struct puntata da ucp, setContext(ucontext_t* ucp) invece riesuma il contesto puntato da ucp, makecontext(ucontext_t *ucp, void (*func)(), int argc, ...) imposta la funzione passatagli come argomento come codice da eseguire una volta avviato contesto puntato da ucp, swapContext(ucontext_t *oucp, ucontext_t *ucp) salva contesto corrente in oucp e riesuma il contesto puntato da ucp.

Memory allocators:

Problemi comuni per la politica di allocazione da scegliere: ci può essere frammentazione ossia c'è abbastanza spazio in totale per soddisfare la richiesta da parte di un processo ma non è contigua! È inoltre importante badare al tempo necessario per soddisfare richieste di get/release. NB per gestire la memoria sono necessarie strutture dati proprie dell'allocatore che sprecano parte della memoria stessa.

SLAB Allocator: si usa in situazioni in cui gli insieme degli oggetti ha una size fissata e/o quando conosci a priori il numero massimo di oggetti che puoi avere. Il buffer di memoria viene suddiviso in Bucket della stessa size. Questo rende molto semplice l'indicizzazione della memoria. Gli unici indirizzi validi sono quelli che puntano all'inizio del bucket di memoria. La richiesta di get può essere eseguita in O(1) attraverso un arrayList di maxSize Int interno allo SLAB. Bisogna tener conto del primo indice dell'array valido (nel caso iniziale l'indice 0), inoltre ogni cella i-esima dell'arrayList conterrà l'indice della cella successiva disponibile. L'ultima cella conterrà il valore -1 (NullIdx). I passaggi della get_Block sono: se l'idx della prossima cella valida è diversa da -1 posso soddisfare la richiesta e calcolo il

puntatore da restituire all'user così (p_buffer_iniziale + idx*size_bucket). I passaggi della release_Block sono: ottenuto il puntatore verifico se la free è già stata effettuata, nel caso lancio un'eccezione, altrimenti aggiorno la cella corrispondente al puntatore nell'arrayList.

Buddy Allocator: si usa quando abbiamo oggetti di size variabile. Si decide a priori la size minima di un bucket di memoria. Ad ogni richiesta di calcola la size del bucket minima consentita che soddisfa la richiesta procedendo ricorsivamente a dividere in due il buffer di memoria. Il "buddy" ottenuto dalla divisione è il fratello del blocco consesso all'user, verrà conservato nella lista dei blocchi liberi riferita ai blocchi di quella size specifica. Le nostre partizioni ricorsive formano un albero binario, possiamo sfruttare le proprietà di questa struttura dati indicizzando ogni BuddyListItem come nodo dell'albero. Ogni BLI avrà il suo livello corrispondente (indice dell'array di liste dei blocchi liberi), un proprio indice, da cui facilmente si può risalire al fratello e al padre del nodo in questione. L'array di liste dei bucket liberi viene gestito tramite uno SLAB Allocator proprio del Buddy Allocator, avremo quindi uno "spreco" di memoria utile a conservare informazioni sulla struttura dell'albero. Alla richiesta di un blocco in primis si calcola la size giusta del blocco da restituire, si controlla nella lista dei blocchi liberi corrispondente se c'è uno libero, nel caso contrario ricorsivamente si vanno a dividere i padri, partendo dal primo libero. Inoltre per facilitare la richiesta di releaseBlock si conserva nei primi 8 bytes il puntatore al BLI riferito al blocco consegnato. In caso di realeseBlock si ottiene il puntatore del BLI conservato nei primi 8 bytes antecedenti il puntatore ottenuto. Nel caso in cui il Buddy del BLI fosse libero si fondono attraverso la loro eliminazione e creando un BLI padre con size la somma dei due bucket fusi. Questo processo è ricorso, si blocca una volta trovato un Buddy allocato o se si arriva a deallocare tutto il buffer di memoria. Per evitare spreco di memoria per conservare i BLI si può utilizzare una bitMap in cui ogni bit indica l'allocazione di ogni nodo dell'albero.

Syscall and Dual Mode:

Interrupts: I moderni sistemi operativi si basano sugli Interrupts. Ogni interazioni con il sistema operativo è triggerata da un interrupt. Gli interrupt sono generati da eventi esterni, eccezioni interne o chiamate esplicite dell'user. Fondamentali perché non risultano bloccanti, bensì la routine di servizio verrà eseguita solo a ricezione dell'interrupt. Alla ricezione di un interrupt il SO salva il contesto corrente e una volta identificata la routine da eseguire in relazione della causa dell'interruzione, la esegue. L'ISR è parte del SO. L'interrupt Vector è un array nelle cui celle sono contenuti i puntatori a funzione della routine da svolgere in corrispondenza dell'interrupt con codice i-esimo. Le eccezioni sono Interrupt generati a software, ne esistono di tre tipi: Trap, generata dopo specifiche istruzioni, Faults prima che determinate istruzioni vengano eseguite, Abort stato in cui un processo non può

essere recuperato (eccezione nell'eccezione). Il segnale di interrupt è implementato ad HW mentre la gestione degli stessi è a carico del SO. Chiamare un INT XX differisce da una CALL YY poichè nella prima il registro FLAGS viene alterato esistono determinati valori che XX può assumere, ossia solo indici del IVT mentre YY è un qualsiasi indirizzo di memoria a cui segue del codice. Data la necessità di prevenire un uso scorretto dell'IVT da parte dell'User esistono 2 differenti modalità: quella User a cui è permesso servirsi di un limitato insieme di istruzioni, e quella Kernel in cui si hanno tutti i privilegi per poter gestire al meglio ogni situazione. La modalità switcha attraverso un bitFlag. Alterarlo però è essa stessa un'istruzione privilegiata perciò si è pensato di "nascondere" tutto l'SO dietro una entry del IVT così che l'User abbia a disposizione un'unica istruzione per invocare il sistema operativo passandogli i parametri o nei registri della CPU o nella stack affinché possa capire quale specifica ISR esso deve svolgere. Le applicazioni User sono così bloccate, potendo chiamare un'unica routine di servizio del SO per gestirle tutte, ad ogni id corrisponderà una specifica syscall. Inoltre il flagBit verrà settato di nuovo a UserMode da l'IRET. Per questioni di portabilità si usano librerie condivise ad alto livello, le quali al loro interno contengono syscall compatibili per ogni SO, potendo così far girare tutti i programmi su qualsiasi piattaforma (Languages Standard). Esistono standard anche per le syscall, ad esempio POSIX, che standardizza il funzionamento di alcune syscall su vari OSes. Si rende necessaria un'architettura a strati per la programmazione affinché si aumenti la portabilità. Più a basso livello si programma e meno portabilità si otterrà per la propria applicazione, più ad alto livello si programma meno sarà efficiente l'applicazione ma ne aumenterà la portabilità.

Processi:

Processo: programma in esecuzione, in un OS multitasking più istanze di uno stesso processo possono essere eseguite concorrentemente. Un processo è caratterizzato dai registri CPU, dalle sue aree di memoria, dalle sue risorse (fd, sem, queues).

Stack Frame: Puntato da un registro speciale (RSP). Conserva il record di attivazione per ogni funzione (argomenti, valore di ritorno, var locali, registri pushati).

Stati Processo: Quando un processo viene creato è nello stato NEW, nel caso fosse pronto per essere schedulato passa nello stato READY. Se schedulato passa nello stato RUNNING. In caso di chiamata I/O passa nello stato di WAITING. Alla terminazione passa in uno stato ZOMBIE fin tanto che il padre o, nel caso orfano, il processo INIT non raccolga il valore di ritorno eliminando il processo in questione.

Fork: System Call usata per creare un nuovo processo. Dopo la fork due istanze dello stesso programma vengono lanciati. La memoria del processo padre è copiata

interamente, compresi file descriptor e altre risorse. Per il figlio il valore di ritorno è 0, per il padre il PID del figlio.

Wait/Waitpid: Syscall bloccante per aspettare la terminazione di uno dei figli (wait) o di uno specifico (waitpid).

Segnali: Sono generati a livello software dal SO o da eventi legati a processi PER IL SO, ossia la CPU non è al corrente della gestione dei segnali, a differenza degli interrupt. Il SO lo gestirà tramite gli handler associati ad ogni segnale. Tramite una struct sigaction si può modificare il comportamento di default legato ad un segnale. Va settato *.sa_handler, la f.ne da svolgere, e impostare i flag che identificano quali altri segnali ascoltare durante lo svolgimento dell'handler. Inoltre con sigaction() si imposta la struct sigaction (handler) in ascolto del segnale identificato da un codice univoco.

Terminazione processo: Un processo padre viene messo a conoscenza della terminazione di un figlio tramite un segnale SIGCHLD. I figli invece alla morte del padre ricevono un segnali SIGHUP (non attivo di default on Linux).

Exit: Un processo termina la sua esecuzione con la exit(return value). Il main chiama implicitamente la exit().

Exec: Rimpiazza tutta la memoria di un processo con quella di un altro caricata da un program file. La memoria del processo precedente viene completamente eliminata insieme a tutte le sue risorse. Versioni: exec(), execv(argv), execve(argv,environ).

Vfork: Se ciò che fa il figlio creato è chiamare EXEC conviene utilizzare una vfork, che si differenzia dalla fork poiché applica la politica SAVE ON COPY, evita cioè di sprecare tempo a copiare alcune parti del processo padre siccome l'exec la dropperà comunque tutta.

PCB: Il Process Control Block è una struttura dati conservata dal Kernel in cui sono contenute informazioni essenziali riguardanti la vita del processo: è presente il suo ID, tutti i puntatori a memoria, il suo stato, lo stato della CPU per il processo in questione, tutte le risorse in uso. Il PCB è salvato in un'area di memoria privilegiata.

Content Switch: Avviene nel momento in cui un processo viene interrotto, dovuto a un interrupt o ad un eccezione. Questo avvia l'esecuzione del codice del kernel. Al ritorno del codice del kernel il processo riesumato potrebbe essere uno qualsiasi tra quelli della lista di ready, dipende dalle decisioni del kernel.

Dettaglio del cs: In primis si salvano nella stack del processo registro FLAGS e IP. Si passa successivamente in Kernel mode per eseguire la giusta ISR. Per poter in futuro riesumare il processo interrotto va salvato il suo completo PCB. Impostiamo poi il registro RSP alla stack del kernel per evitare di sporcare quella del processo.

Ora si può completare l'esecuzione dell'ISR. Infine si sceglie il successivo processo RUNNING e si imposta il registro RSP alla stack del processo scelto. Bisogna infine ristabilire il contesto del processo selezionato attraverso le informazioni contenute nel PCB. La IRET finale del Kernel code ristabilirà il registro FLAGS e riprenderà l'esecuzione del processo dall'istruzione puntata da IP.