**MINIX3**

Uno degli obiettivi principali di Minix 3 è mantenere il kernel il più piccolo e semplice possibile per garantire l'affidabilità e la manutenibilità. Questa è una delle ragioni per cui Minix 3 evita l'uso di semafori e di altri meccanismi di sincronizzazione complessi.

L'uso di semafori e altri meccanismi di sincronizzazione può portare a problemi di concorrenza e deadlock, ed è più difficile da testare e mantenere. Minix 3 cerca di minimizzare questi problemi mantenendo il kernel il più semplice possibile ed invece di semafori, utilizza un meccanismo basato su messaggi per la comunicazione tra processi, che è più prevedibile e meno suscettibile a problemi di sincronizzazione.

Tuttavia, è importante notare che questa scelta di design ha i suoi trade-off. Minix 3 potrebbe non essere adatto per tutti gli utilizzi, soprattutto quelli che richiedono un alto grado di prestazioni e scalabilità o che richiedono la gestione di carichi di lavoro altamente concorrenti.

Sono fornite tre primitive per scambiare messaggi, chiamate secondo le librerie C:

* send(dest,&message) invia un messaggio a dest
* receiver(source,&message) per ricevere un messaggio da source o ANY
* sendrec(src\_dest,&message) manda un messaggio e aspetta per la risposta dallo stesso processo

Il secondo parametro nelle call è il local address dei dati da scambiare. Il meccanismo di scambio dei messaggi viene effettuato dal kernel che invia i messaggi dal sender al receiver. La risposta( in sendrec) sovrascrive il messaggio originale.

Quando un processo invia un messaggio ma il destinatario non è ancora pronto a riceverlo esso si blocca finchè la destinazione non effettua una *receive*

In altre parole, MINIX 3 utilizza il metodo del rendezvous per evitare i problemi legati all'invio di messaggi che sono stati messi in coda ma non sono ancora stati ricevuti. Il vantaggio di questo approccio è che è semplice ed elimina la necessità di gestire i buffer (compresa la possibilità di esaurire i buffer). Inoltre, poiché tutti i messaggi hanno una lunghezza fissa determinata in fase di compilazione, gli errori di sovraccarico del buffer, una fonte comune di bug, vengono prevenuti in modo strutturale.

Message Types:

• 7 tipi di messaggi (vedi file ipc.h)

– Definisce la struct message, e send, receive, sendrecv

• il messaggio sarebbe potuto essere un array di bytes (non strutturato)

– Minix usa una union sui tipi di messaggio

– 7 tipi: vanno da 1 to 8 (6 è obsoleto)

• Message struct: m\_source: il mittente

» m\_type: tipo del messaggio

» campi dato

– Tecnicamente, x.m\_u: union di struct, x.m\_u.m\_1.m1\_i1

• Macro per semplicità: x.m1\_i1

• tipi di dato: integer, long, pointer, char, char array

I sette tipi di messaggi utilizzati in MINIX 3. Le dimensioni degli elementi dei messaggi varieranno a seconda dell'architettura della CPU; la figura mostra punti a 32 bit per Intel.

Immagine che contiene testo, numero

Descrizione generata automaticamente

Come vengono usati i messaggi?

– Chiamate Kernel

– proc.c sys\_call

• converte software interrupt in message, send e recv

• lavoro effettivo svolto in mini\_send, mini\_recv, mini\_notify

Il codice di alto livello per la comunicazione tra processi si trova in proc.c. Il compito del kernel è tradurre sia un'interruzione hardware che un'interruzione software in un messaggio. Le prime sono generate dall'hardware e le seconde sono il modo in cui una richiesta di servizi di sistema, cioè una chiamata di sistema, viene comunicata al kernel. Questi casi sono sufficientemente simili da poter essere gestiti da una singola funzione, ma è stato più efficiente creare funzioni specializzate.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Esiste un'altra importante primitiva di passaggio di messaggi. È chiamata dalla procedura della libreria C.

notify(dest);

e viene utilizzata quando un processo deve rendere consapevole un altro processo di qualcosa di importante è accaduto. Una notifica non è bloccante, il che significa che il mittente continua a eseguire il proprio codice indipendentemente dal fatto che il destinatario sia in attesa o meno. Poiché non blocca, una notifica evita la possibilità di un deadlock dei messaggi. Il meccanismo dei messaggi viene utilizzato per consegnare una notifica, ma le informazioni trasmesse sono limitate. Nel caso generale, il messaggio contiene solo l'identità del mittente e un timestamp aggiunto dal kernel. A volte è necessario solo questo

Ogni processo di sistema ha una bitmap per le notifiche in sospeso, con un bit distintivo per ogni processo di sistema. Quindi, se il processo A ha bisogno di inviare una notifica al processo B in un momento in cui il processo B non è bloccato su una ricezione, il meccanismo di passaggio dei messaggi imposta un bit corrispondente ad A nella bitmap delle notifiche in sospeso di B. Quando B esegue finalmente una ricezione, il primo passo è controllare la sua bitmap delle notifiche in sospeso. In questo modo può apprendere le notifiche tentate da più fonti. Il singolo bit è sufficiente per rigenerare il contenuto informativo della notifica. Esso indica l'identità del mittente, e il codice di passaggio dei messaggi nel kernel aggiunge il timestamp quando la notifica viene consegnata. I timestamp sono utilizzati principalmente per verificare se sono scaduti dei timer.

**OS161**

OS/161 è un sistema operativo educativo utilizzato in ambienti accademici per insegnare i principi dei sistemi operativi. Nella progettazione e nell'implementazione di OS/161, vengono introdotti e studiati vari meccanismi di sincronizzazione, che sono fondamentali per comprendere come i sistemi operativi gestiscono la concorrenza e il coordinamento tra i processi. Di seguito, sono elencati alcuni dei meccanismi di sincronizzazione comuni in OS/161:

1. \*\*Semafori:\*\* OS/161 include un'implementazione dei semafori come meccanismo di sincronizzazione. I semafori vengono utilizzati per coordinare l'accesso concorrente a risorse condivise. Ci sono semafori binari (che possono essere bloccati o sbloccati) e semafori generali (che tengono traccia di un conteggio).

2. \*\*Locks (bloccaggi):\*\* I bloccaggi (locks) sono utilizzati per garantire l'accesso esclusivo a una risorsa condivisa. I lock sono spesso implementati come semafori con un valore iniziale di 1 (semafori binari) e vengono utilizzati per proteggere le sezioni critiche del codice.

3. \*\*Condition Variables (variabili di condizione):\*\* Le variabili di condizione vengono utilizzate in combinazione con i lock per consentire l'attesa e il risveglio di processi in base a determinate condizioni. Ad esempio, possono essere utilizzate per implementare la sincronizzazione tra produttori e consumatori in un problema di buffer circolare.

In OS/161, gli studenti e i ricercatori possono sperimentare con questi meccanismi di sincronizzazione per acquisire familiarità con i concetti fondamentali della gestione della concorrenza nei sistemi operativi. Ogni meccanismo ha le sue applicazioni e i suoi casi d'uso specifici, e la scelta del meccanismo dipenderà dalle esigenze del problema specifico che si sta cercando di risolvere.