

sistemi operativi riassunto

Matteo

January 14, 2026

1 processi

Def processo: un attività controllata da un programma che si svolge su un processore.

programma e un entità statica, un processo è dinamico.

il processo è l'attualizzazione del programma.

assioma del "finite process": ogni processo viene eseguito a una velocità finita, costante ma sconosciuta.

Ad ogni istante, un processo può essere totalmente descritto dalle seguenti componenti:

- La sua immagine di memoria, divisa in:
 1. la memoria assegnata al processo (ad es. testo, dati, stack)
 2. le strutture dati del S.O. associate al processo (ad es. file aperti)
- la sua immagine nel processore: contenuto dei registri generali e speciali
- il suo avanzamento del processo: descrive lo stato corrente del processo: ad esempio, se è in esecuzione o in attesa di qualche evento

più processi possono eseguire lo stesso programma stato dei processi:

- running
- waiting: in attesa di qualche evento
- ready: è pronto ma il processore è impegnato in qualcos'altro

2 introduzione concorrenza

gestione di processi multipli:

- Multiprogramming: più processi su un solo processore parallelismo apparente
- Multiprocessing: più processi su una macchina con processori multipli parallelismo reale
- Distributed processing: più processi su un insieme di computer distribuiti e indipendenti parallelismo reale

concorrenza: due programmi vengono eseguiti insieme con parallelismo reale o apparente
molte operazioni possono essere create come un insieme di thread concorrenti

interleaving: per ogni processore al massimo un processo alla volta e attivo, multipli processi sono alternati nel tempo

overlapping: più processi sono eseguiti contemporaneamente su più processori, sono alteranti nello spazio

multiprogramming e multiprocessing condividono gli stessi errori derivanti dal fatto che non è possibile predire gli istanti temporali in cui avvengono le operazioni e i processi accedono a risorse condivise

se il risultato di una programmazione concorrente dipende dalla temporizzazione dei processi, si dice che quella programmazione contiene una RACE CONDITION

è necessario eliminare la race condition scrivere programmi concorrenti è più difficile che scrivere programmi sequenziali perché errori possono anche venire da errori nelle iterazioni tra loro

esistono processi concorrenti “ignoranti l’uno dell’altro”, sono indipendenti ma concorrono per le stesse risorse, il sistema deve arbitrarle e sincronizzarle. questi non sono creati per lavorare insieme

esistono processi indirettamente a conoscenza uno dell’altro che non si conoscono per id ma interagiscono indirettamente tramite le risorse condivise, anche loro devono essere coordinate e sincronizzate

poi ci sono processi direttamente a conoscenza uno dell’altro per id, che comunicano direttamente tramite messaggi, cooperano per certi scopi e informano anche altri processi, il sistema deve fornire meccanismi di comunicazione

la memoria può essere condivisa o privata

2.1 proprietà della concorrenza

Una proprietà di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso

esistono 2 tipi di proprietà:

- safety: (nothing bad happens), l’obiettivo è che se il programma va avanti vada nella direzione giusta
- liveness: (something good eventually happens), il programma non si ferma mai

nei programmi concorrenti safety significa evitare interferenza tra processi nelle risorse condivise

liveness significa che la sincronizzazione non deve impedire al programma di andare avanti, cioè non è ammissibile che un processo debba attendere indefinitamente

2.2 problemi della concorrenza

l’accesso ad una risorsa si dice mutualmente esclusivo se ad ogni istante, al massimo un processo può accedere a quella risorsa

questo risolve il problema della non interferenza ma può creare quello del deadlock, cioè avviene quando dei processi si bloccano a vicenda per un motivo o per l’altro, ad esempio il processo è bloccato perché ha bisogno di usare due processi contemporaneamente ma

la mutua esclusione lo previene, o due processi hanno bisogno di una risorsa occupata dall'altro.

non interferenza: i processi non devono alterare l'esecuzione degli altri processi in modo non voluto.

starvation invece e quando un processo e fermo perche ha bisogno di una risorsa ma altri processi continuano a saltargli davanti nella coda per la risorsa

azioni atomiche: compiute in modo indivisibile, soddisfano la condizione o tutto o niente, non interferisce con altri processi durante la sua esecuzione

certe azioni vanno eseguite in modo atomico per garantire la non interferenza

le parti di un programma che utilizzano una o piu risorse condivise vengono dette sezioni critiche

3 gestione problemi

3.1 intro

obbiettivi:

- usare solo azioni atomiche nelle sezioni critiche
- evitare blocchi(deadlock,starvation)
- evitare attese inutili, si fa aspettare un processo solo se gli serve una risorsa sezione critica occupata

il sistema non capisce necessariamente da solo cosa e una sezione critica e cosa no quindi devi programmare i processi accordingly (es: se entri in sezione $x=x++$, e per entrare x deve essere 0) quando entri in una sezione critica devi rispettare 4 proprieta:

- mutua esclusione: max 1 processo attivo che deve accedere a quella risorsa
- assenza di deadlock
- assenza di delay non necessari: Un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo
- eventual entry(assenza di starvation): ogni processo che lo richiede prima o poi accede alla risorsa

se un processo entra in una sezione critica, prima o poi ne uscirà, quindi un processo non può terminare se non fuori da una CS(critical section) si può provare a gestire i CS tramite:

1. software: cioè si programmano i processi apposta, rischi di fare errori ed e probabile avere busy waiting
2. hardware: utilizzano istruzioni speciali del linguaggio, progettate apposta, e efficienti ma non general purpose
3. ci sono poi approcci basati sul S.O. o sul linguaggio di programmazione(es: semafori o monitor)

3.2 soluzioni software

algoritmo di dekker

algoritmo di peterson (più semplice)

sono entrambe soluzioni software, funzionano ma sono entrambe basate sul busy waiting, che non andrebbe usato.

busy waiting: controllo iterativo di una condizione di accesso a una CS, così il processo rimane attivo ma non fa nulla di utile

3.3 soluzioni hardware

altrimenti si possono usare soluzioni hardware, cioè usare istruzioni speciali per l'hardware

esempio: si possono disabilitare gli interrupt in entrata alla CS e riabilitarli dopo

gli interrupt sono ciò che danno il segnale per alternare i processi, senza e fisicamente impossibile non avere mutua esclusione

problemi:

- danneggia il parallelismo
- si lascia ai processi la responsabilità di riattivare gli interrupt che può generare errori
- ha senso solo su uniprocessori

Le (quasi) sezioni critiche realizzate con istruzioni speciali vengono chiamate spinlock in generale, i processi hardware:

- possono generalmente essere applicati qualsiasi numero di processi sia su sistemi mono che multiprocessori
- semplice da verificare
- può essere utilizzato per sezioni critiche multiple
gli svantaggi sono
- c'è ancora busy waiting
- non elimina problemi di starvation
- è difficile da programmare

noi invece vorremmo paradigmi poco complessi e facili da implementare.

4 semafori

per avere un paradigma semplice ed efficace, si pensa ai semafori:

due o più processi si mandano segnali in modo che un processo rimanga bloccato in un punto specifico fino a quando non riceve un altro segnale

vi sono due operazioni:

v: usata per inviare il segnale

p: usata per attendere il segnale

il semaforo può essere visto come una variabile intera inizializzato a un numero non negativo

p attende che il semaforo sia positivo e lo decrementa

v aumenta il valore del semaforo

v e p sono entrambe atomiche

struttura dati è a coda e ha una politica FIFO, senza si potrebbe avere starvation

non abbiamo eliminato il busy waiting ma lo abbiamo limitato a dentro le operazioni v e p.

4.1 semafori binari

un semaforo binario è un semaforo che può assumere solo i valori 0 e 1, può essere usato per garantire la mutua esclusione.

ha lo stesso potere espressivo di un semaforo normale.

4.2 Problemi con i semafori

i semafori non garantiscono l'assenza di deadlock e starvation, e compito del programmatore evitare questi problemi

ecco alcuni problemi classici:

- produttore-consumatore
- buffer limitato
- filosofi a cena
- lettori-scrittori

passi secondo Andrews per risolvere questo tipo di problemi:

1. definire il problema, identificare i processi, specificare i problemi di sincronizzazione, introdurre le variabili necessarie e definire un'invariante
2. abbozzare una soluzione, produrre un primo schema di soluzione, e identificare le regioni che richiedono accesso atomico o mutualmente esclusivo
3. garantire l'invariante, controllare che l'invariante sia sempre verificato
4. implementare le azioni atomiche, esprimere le azioni atomiche e gli statement await utilizzando le primitive di sincronizzazione disponibili

essendo i semafori costruiti di basso livello, è facile commettere errori (ad es. scambiare di ordine P e V), per questo esistono costrutti di più alto livello come i monitor

5 monitor

monitor consiste in

- dati locali
- una sequenza di inizializzazione

- una serie di procedure che operano sui dati locali

le caratteristiche principali sono:

- i dati locali sono accessibili solo alle procedure del modulo stesso
 - un processo entra in un monitor invocando una delle sue procedure
 - solo un processo alla volta può essere all'interno del monitor; gli altri processi che invocano il monitor sono sospesi, in attesa che il monitor diventi disponibile
- sono costruiti un come gli oggetti in programmazione ad oggetti:

- codice di inizializzazione= costruttore
- procedure entri= metodi pubblici(sono richiamabili dall'esterno)
- dati locali= attributi privati
- procedure interne= metodi privati

il monitor fornisce un meccanismo di mutua esclusione, e inoltre ha bisogno di un meccanismo di sincronizzazione

5.1 sincronizzazione del monitor

esiste la variabile di condizione C, che ha due operazioni:

- **wait(C)**: viene rilasciata la mutua esclusione e il processo che la invoca viene sospeso e messo in attesa sulla coda di C
- **signal(C)**: se c'è un processo in attesa su C, uno di questi viene risvegliato e messo nella coda del monitor secondo FIFO, il chiamante viene sospeso e verrà riattivato al rilascio della mutua esclusione.

c.wait e c.signal sono diverse da p e v.

signal non ha effetto se non ci sono processi in attesa, wait è sempre bloccante, e il processo risvegliato da signal viene eseguito per primo.

5.2 politiche di segnalazione

una politica di signaling determina cosa succede quando un thread fa signal su una condition variable ne esistono diverse:

- **signal urgent**: il thread che fa signal viene sospeso e il thread risvegliato prende la mutua esclusione, dobbiamo usare questo in questo corso.
- **signal and wait**: il thread che fa signal continua l'esecuzione e il thread risvegliato viene messo in coda per la mutua esclusione
- **signal and return**: il thread che fa signal continua l'esecuzione e il thread risvegliato prende la mutua esclusione
- **signal and continue**: il thread che fa signal continua l'esecuzione e il thread risvegliato viene messo in coda per la mutua esclusione

slide 164: se value =0 allora wait, altrimenti il processo si esegue e value – se signal, il processo chiamante si sospende e viene risvegliato un processo in attesa che riprende da subito dopo dove si era fermato

6 message passing

nei semafori e monitor, la comunicazione avviene tramite memoria condivisa

il meccanismo detto: **message passing** è un paradigma di comunicazione tra processi, la sincronizzazione avviene tramite lo scambio di messaggi e non semplice segnalazione

un messaggio è un insieme di dati formattati dal mittente e interpretati dal ricevente

un meccanismo di "scambio messaggi" copia le informazioni di un messaggio da uno spazio di indirizzamento di un processo allo spazio di indirizzamento di un altro processo esistono le operazioni:

- send: manda il messaggio, il destinatario deve essere specificato
- receive: riceve il messaggio, il mittente può essere specificato

il passaggio dallo spazio di indirizzamento del mittente a quello del destinatario è mediato dal sistema operativo (protezione memoria)

diversi tipi di message passing(MP):

1. MP sincrono

- send sincrono
- receive bloccante

2. MP asincrono

- send asincrono
- receive bloccante

3. MP completamente asincrono

- send asincrono
- receive non bloccante

6.1 MP sincrono

sintassi send: `void (ssend(msg_t m, pid_t dst))`

Il mittente spedisce il messaggio m al processo dst, restando bloccato fino a quando dst non riceve il messaggio

sintassi receive: `msg_t sreceive(pid_t snd)`

il destinatario riceve un messaggio dal processo snd; se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio

il destinatario può non essere specificato.

6.2 MP asincrono