

Processi e Thread

Durante l'esecuzione di un programma utente, le attività dell'elaboratore sono molteplici e alcune attività riguardano: la lettura da disco, la stampa di informazioni su terminale o su stampante, etc...

In un sistema **Multitasking**, la CPU passa da un programma all'altro dedicando a ciascuno di essi decine o centinaia di millisecondi. Ovviamente nell'arco temporale di un secondo la CPU esegue diversi processi (programmi) creando l'impressione di eseguirli parallelamente (pseudo-parallelismo).

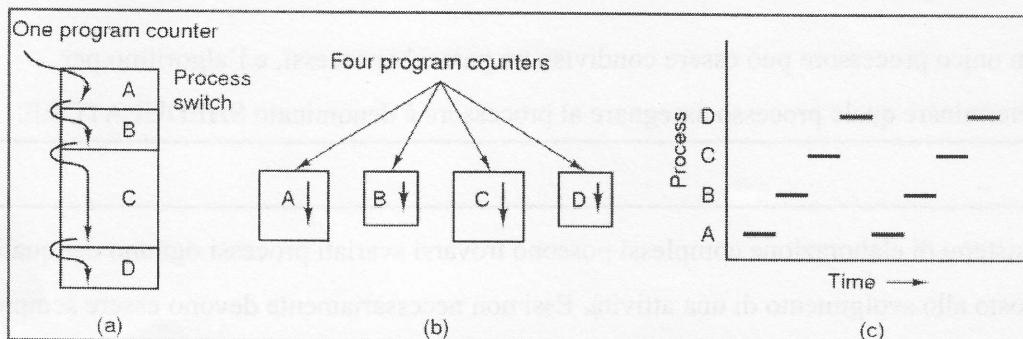
L'uso concorrente della CPU è di dificile formalizzazione e il modello che tiene traccia di tutti i programmi "parallelamente" eseguiti dalla CPU è basato sul concetto di Processo.

In sintesi un processo non è altro che una collezione di informazione contenente: un programma in esecuzione, i suoi valori correnti, il PC (Program Counter), i registri, le variabili, etc.

Inoltre ogni processo dispone di una sorta di CPU virtuale a cui fa riferimento.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Processi



- Processi con un solo PC;
- Processi con diversi PC logici caricati di volta in volta nel PC reale;
- Evoluzione Temporale dei Processi.

L'andamento con cui un processo svolge i suoi calcoli non si uniforma alla metodologia con cui la CPU passa da un processo all'altro. Probabilmente la sequenzialità delle azioni del S.O. su di un processo risulterebbe diversa, anche in presenza degli stessi processi concorrenti, se fosse rimandato in esecuzione in tempi diversi.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Esempio (**Automobilista Alla Guida di un Veicolo**):

Immaginiamo di osservare un automobilista medio durante la sua guida, esso è impegnato a osservare il contesto esterno per evitare i pericoli che lo circondano ed il contesto interno per controllare il giusto funzionamento del veicolo.

Al semaforo l'automobilista non sarà impegnato nella guida ma sarà impegnato ad evitare i lavavetri o le invasioni nel veicolo dei venditori ambulanti, a leggere il giornale, a parlare al cellulare.

Il guidatore è passato da un contesto di guida a altri tipi di contesti totalmente diversi dal precedente.

Identificando il guidatore con la CPU il cui processo iniziale è la guida e i cui dati di input provengono dall'ambiente esterno e interno (dati nuovi o elaborati). I successivi processi quando il guidatore è fermo al semaforo sono:

Processo “lavavetro”, dati di ingresso: l'omino che si spalma sul parabrezza; dati di uscita: attivare i tergivetri;

Processo “leggi giornale”, dati d'ingresso: provenienti dal dispositivo cartaceo, la borsa va giù; dati d'uscita: realizza di avere perso soldi;

Processo “venditore ambulante”, dati d'ingresso: ortaggi, calcolo sulla qualità del prodotto; dati uscita: compro/noncompro.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

60

In questo caso, l'idea chiave per identificare i componenti di un processo sta nell'attività che deve compiere, nei suoi dati in I/O e nel suo stato.

Un unico processore può essere condiviso tra parecchi processi, e l'algoritmo per determinare quale processo assegnare al processore è denominato SHEDULATORE.

Nei sistemi di elaborazione complessi possono trovarsi svariati processi ognuno dei quali è preposto allo svolgimento di una attività. Essi non necessariamente devono essere sempre attivi, e possono essere creati al bisogno e successivamente eliminati quando ha termine il loro compito.

Quattro eventi principali provocano la creazione di un processo(foreground, background):

- 1) L'inizializzazione del sistema.
- 2) L'esecuzione di una chiamata di sistema per la creazione di un processo, effettuata da un processo in esecuzione.
- 3) Una richiesta da parte dell'utente per creare un nuovo processo.
- 4) L'inizio di un job batch.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

61

Processi in foreground e background

L'inizializzazione del sistema - All'attivazione di un sistema operativo vengono generalmente creati diversi processi alcuni dei quali vengono eseguiti in foreground, generalmente interagenti con gli utenti, altri ancora in background che hanno specifiche funzioni o servizi.

I processi in background sono generalmente dormienti e si attivano al verificarsi di un evento: arrivo di una email, distribuzione di pagine web, stampe, ect.

In generale tali processi prendono il nome di demoni (deamon) e nei sistemi di grandi dimensioni ve ne sono a dozzine.

L'esecuzione di una chiamata di sistema - Ai processi creati all'avvio del sistema possono affiancarsi processi creati successivamente sia dagli utenti sia dal S.O.. Essi possono invocare chiamate al sistema per chiedere aiuto a svolgere i propri compiti (e.i. creazione di altri processi).

Richiesta da parte dell'utente - Nei sistemi interattivi, gli utenti possono lanciare un programma scrivendo il suo nome o facendo doppio click sull'icona che lo rappresenta, in entrambi i casi vengono creati processi per l'esecuzione di un applicativo.

62

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

L'inizio di un job batch - I processi batch, che si trovano sui grossi mainframe, sono utilizzati dagli utenti per accodargli i propri processi da elaborare in background. Tali processi vengono eseguiti quando il S.O. si rende conto di avere sufficienti risorse per elaborarne uno di essi. Il primo job della coda batch viene prelevato per la sua esecuzione.

Tecnicamente viene creato un nuovo processo tramite l'esecuzione di un'apposita chiamata di sistema, effettuata da uno dei seguenti processi presenti nel sistema:

- un processo utente;
- un processo di sistema lanciato dal device di Input (tastiera, mouse,...);
- un processo che gestisce l'esecuzione dei batch.

Analiticamente il processo esegue una chiamata di sistema

per la creazione di un nuovo processo e indica

direttamente o indirettamente quale APPLICATIVO deve essere eseguito.

63

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Sia in Unix che in Windows, dopo la creazione di un processo, il genitore ed il figlio hanno **spazi di indirizzamento distinti** e le eventuali modifiche apportate su ciascuno di essi non sono visibili all'altro.

Comunque è possibile che un processo appena creato **condivida alcune risorse** del proprio creatore, come i file aperti.

64

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

La Terminazione dei Processi

La terminazione di un processo può essere imputata ad una delle seguenti cause:

- 1) Terminazione normale (volontaria).
- 2) Terminazione con Errore warning (volontaria).
- 3) Si è verificato un Errore fatale (involontario).
- 4) È stato ucciso (“kill”) da un altro processo(volontario/involontario).

Terminazione normale - La maggior parte dei processi termina perché ha finito il proprio lavoro; quando il processo termina il proprio compito esegue una chiamata (EXIT) di sistema per avvertire il S.O. della sua terminazione.

Terminazione con Errore warning - in tale condizione il processo individua un errore e termina la sua esecuzione.

Ad esempio nei programmi interattivi, quando l'utente inserisce parametri errati, **generalmente** NON si hanno terminazioni normali.

65

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Vi è verificato un errore fatale - Tale terminazione molte volte è causata da un errore di programmazione (esecuzione di istruzioni illegali, riferimenti a parti di memoria che non esistono o non permesse, divisione per 0,etc). Soluzioni alternative possono essere adottate affinché il processo non termini al verificarsi dell'errore ma esso viene semplicemente interrotto al verificarsi dell'errore fatale (manipolazione delle eccezioni).

È stato ucciso - Tale terminazione si verifica quando un processo invoca una chiamata di sistema per l'eliminazione di un altro processo (KILL), in qualsiasi caso il processo che decide l'eliminazione deve avere i privilegi per poterlo fare.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

66

Gerarchie di Processi

La creazione di un processo da parte di un altro processo implica una relazione tra di essi, anche rimanendo indipendenti, ed in particolare il processo creato risulta essere figlio del processo creante.

Il processo figlio può creare a sua volta altri processi figli, formando una gerarchia di processi.

La gerarchia dei processi è estremamente importante poiché possono essere inviati dati ad un gruppo di processi aggregati per gerarchie.

Esempio: I comandi impartiti dalla tastiera o da una finestra sono inviati a tutti i processi che sono stati gerarchicamente creati e associati ad essi. Ovviamente ciascun processo è assolutamente in grado di catturare i dati e decidere in totale indipendenza cosa fare.

Nota: In UNIX tutti i processi nell'intero sistema appartengono al processo INIT.

Esso è colui che genera il processo di interfaccia con l'Utente - LOGIN; è attraverso il processo di login che si genera il processo di SHELL, quindi tutti i processi sono discendenti del processo INIT.

In Windows i processi sono orizzontali, non esiste una gerarchia se non quella creata dall'utente, ma anche in questo caso il processo padre può passare il PID del processo figlio (Handle) ad un altro processo, vanificando il concetto di gerarchia.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

67

Gli stati dei Processi

I processi pur essendo delle entità indipendenti, con un proprio contatore di programma e un proprio stato interno, possono o devono interagire con gli altri processi per lo scambio dei dati. Questo fenomeno comporta l'assegnazione di uno stato ad un processo per conoscere la sua condizione istante per istante.

Se due processi si devono scambiare dati e il primo processo non è ancora pronto per fornirli al secondo quest'ultimo deve bloccarsi sino a quando il primo processo non è pronto per rilasciare i dati.

ESEMPIO: `cat cp1 cp2 cp3 | grep albero`

il comando `grep` di ricerca della stringa "albero" all'interno dei file (cp1, cp2, cp3) concatenati dal comando `cat` è bloccato sino a quando il `cat` non finisce la sua operazione.

È anche possibile che un processo concettualmente pronto si blocchi, non per mancanza di dati ma perché il S.O. ha deciso di allocare, per un certo periodo di tempo, la CPU ad un altro processo.

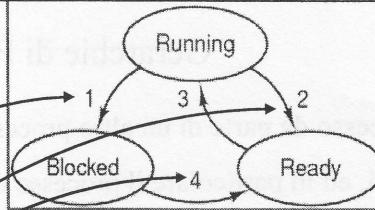
Motivi di Sospensione	
Mancanza di dati da elaborare	Non assegnazione della CPU da parte del S.O.

68

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Stati di un Processo

- 1) In esecuzione
- 2) Pronto
- 3) Bloccato



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

In esecuzione (running) - quando il processo si trova in questo stato ha il possesso della CPU.

Pronto (ready) - può andare in esecuzione e prendere possesso della CPU, esso successivamente può essere sospeso temporaneamente per permettere ad altri processi pronti di prendere possesso della CPU.

Bloccato (blocked) - impossibilitato ad andare in esecuzione in quanto è in attesa di un qualche evento esterno.

Transizioni Possibili:

- 1) Il processo si blocca, è in attesa di un dato di input o di una risorsa.
- 2,3) Tali transizioni sono provocate dallo SCHEDULATORE per distribuire la CPU ai Processi.
- 4) Questa transizione si verifica quando il processo bloccato riceve la risorsa e può quindi concorrere per la CPU.

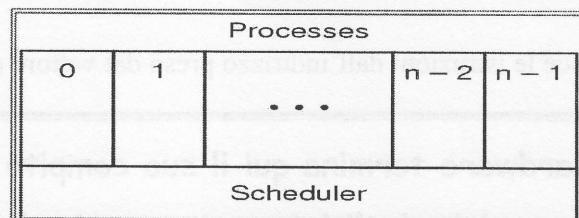
69

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Considerazioni Generali sul modello a Processi

- Alcuni processi eseguono programmi che rispecchiano i comandi dati dall'utente.
- Altri processi sono parte del sistema e gestiscono compiti come ad esempio l'esecuzione dei servizi ai file sui dischi o nastri.
- Anche quando la risorsa è disponibile il sistema può decidere di bloccare il processo che è Running sulla CPU per assegnarla ad un nuovo processo

In definitiva il S.O. può essere modulato sul modello dei processi, in cui i processi sono attivi o bloccati se secondoché la risorsa da essi richiesta è disponibile o no.



NOTA: Il Trattamento delle interruzioni e dell'effettiva attivazione e blocco dei processi è mascherato dallo **Schedulatore**.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

70

Implementazione del Modello a Processi

La realizzazione del modello a processi si ottiene attraverso una tabella dei processi (Process Table) nella quale il S.O. inserisce le strutture dei processi.

Ciascuna struttura, una per ogni processo, ha al suo interno informazioni su:

- lo stato del processo
- il suo program counter
- lo stack pointer
- l'allocazione della memoria
- lo stato dei suoi file aperti
- le informazioni necessarie per l'addebito dei costi e per la schedulazione
- qualsiasi altra informazione utile per far ripartire il programma da dove era stato interrotto come se non si fosse mai fermato.

NOTA: nella tabella dei processi NON è presente lo spazio dell'indirizzamento.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

71

Multitasking con una macchina con una sola CPU “Interrupt”

Ad ogni classe di dispositivi (risorse) viene associata una locazione di memoria spesso vicina alla parte bassa della memoria, chiamata vettore delle interruzioni (interrupt vector) che contiene l'indirizzo in cui è posta la procedura per la gestione dello specifico interrupt.

Supponiamo che una risorsa abbia mandato un segnale di Interrupt, l'hardware dedicato alla gestione degli Interrupt esegue i seguenti passi:

1. Il program counter del processo running, la sua parola di stato e i registri utilizzati vengono messi sullo stack corrente;
2. La CPU acquisisce le istruzioni dall'indirizzo preso dal vettore dell'interrupt.

L'interrupt hardware termina qui il suo compito lasciando spazio alla procedura di servizio degli interrupt propria del S.O. (software)

3. La procedura di gestione dell'interrupt svuota i registri della CPU, rende libero il PC mettendo tali informazioni nella struttura del processo corrente della *Process Table*.

72

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Thread

Nei tradizionali sistemi operativi ad ogni processo è associato un solo **Thread** di esecuzione (filo di esecuzione) ma in molti casi è opportuno avere più thread associati ad un processo per il controllo dello stesso spazio di indirizzamento.

I thread lavorano in parallelo eccetto che per i riferimenti allo spazio di indirizzamento.

In questo contesto un processo può essere visto come un catalizzatore di risorse tra loro correlate (la correlazione è la soluzione al problema che il programma tenta di risolvere).

Ossia un Processo è composto da

- Uno spazio di indirizzamento (testo programma e dati);
- Risorse (file aperti, processi figli, allarmi, gestore di segnali, informazioni);

Tale composizione di informazioni, raggruppata sotto forma di processo, può essere gestita più facilmente.

73

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Thread

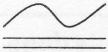
Il Thread di esecuzione o più semplicemente Thread, racchiuso in ogni processo, ha:

- un PC (Program Counter) che tiene traccia della prossima istruzione da eseguire;
- dei registri per mantenere le variabili di lavoro correnti
- uno stack che contiene la storia dell'esecuzione, un record per ogni procedura chiamata e non ancora chiusa.

Processi e Thread sono delle entità distinte e possono essere pensate separatamente.

(NOTA: non può esistere un thread senza il suo processo ma possono esistere più thread associati allo stesso processo, quindi il numero dei thread presenti in un S.O. è \geq al numero dei processi.)

In tal senso è possibile avere più thread di esecuzione sullo stesso ambiente di processo che in larga misura sono indipendenti l'uno dall'altro.

Più Thread afferenti allo stesso processo		Più Processi in esecuzione "parallela" sullo stesso calcolatore.
---	---	--

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

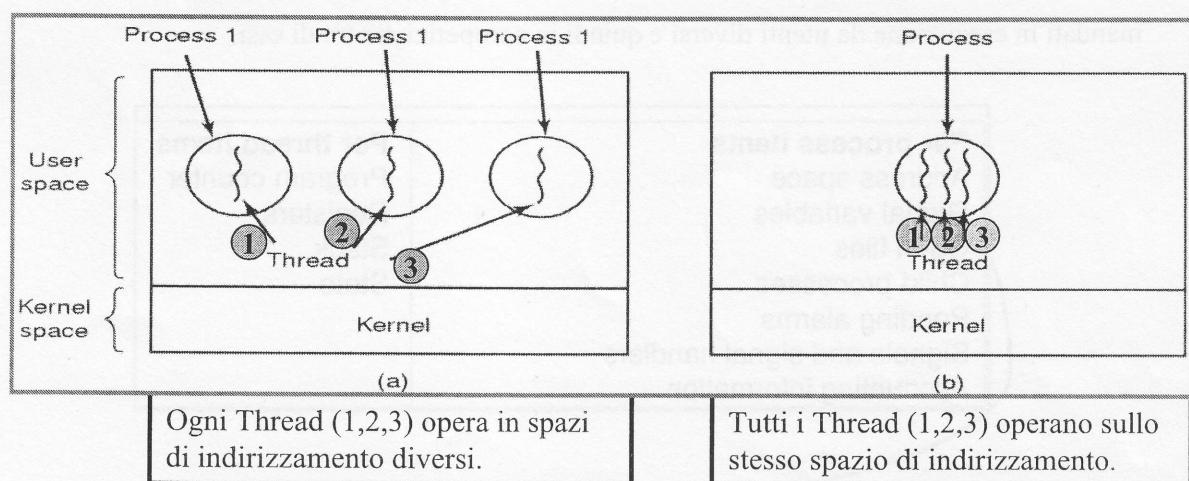
74

Più **Thread** di esecuzione (MultiThreading) su uno stesso processo **condividono**:

Spazio di indirizzamento, i file aperti, risorse in generale.

Più **Processi** in esecuzione parallela sullo stesso calcolatore **condividono**:

Memoria fisica, dischi, stampanti, risorse in genere.



<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

75

La gestione della CPU, nel caso di sistemi MultiThread, è simile al caso in cui si hanno più processi sequenziali e la CPU viene condivisa tra di essi in modo da farli apparire paralleli.

I Thread in un processo non sono indipendenti come lo sono invece i processi distinti di un unico programma.

Perché?

- 1) Hanno lo stesso spazio di indirizzi;
- 2) Condividono le risorse del processo;
- 3) Fanno riferimento alle stesse variabili globali.

Poiché ogni Thread può accedere ad ogni indirizzo di memoria nello spazio di indirizzamento del processo, un Thread può scrivere, leggere, cancellare lo stack di un altro Thread dello stesso processo:

- Non esiste protezione tra i THREAD,
- È impossibile da realizzare,
- Non dovrebbe essere necessaria.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

76

La protezione, tra i Thread di esecuzione di un processo, non ha motivo di esistere in quanto l'utente che ha creato un processo MultiThread, lo ha creato in modo da fare cooperare i thread e non per renderli antagonisti.

Nei Processi ciò non accade in quanto processi di uno stesso programma possono essere mandati in esecuzione da utenti diversi e quindi in competizione tra di essi.

Per process items
Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

Per thread items
Program counter
Registers
Stack
State

Quindi se un thread di un processo apre un file, questo può essere letto o scritto da tutti i Thread di quel processo poiché il File è una risorsa del processo e quindi comune a tutti i Thread

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

77

Esempio d'uso dei Thread

Realizzazione di un Test editor:

- **MONO-Thread:** L'edit di un grosso volume implica in un dato istante di dover cancellare la prima riga e successivamente spostarsi alla pagina 600, questo implica che l'editor dopo la cancellazione dovrà riformattare il testo sino alla pagina 600 con un enorme tempo di elaborazione, questo nel caso in cui un unico Thread è associato al processo di editing.
- **MULTI-Thread:** Di contro supponiamo di avere 2 Thread, uno per la gestione dell'input/output e l'altro per la gestione della formattazione, dopo la cancellazione della prima riga, utilizzando il thread 1 di interfaccia, parte il thread 2 per la formattazione in background senza interrompere il thread 1. Se si è fortunati, l'utente manipolerà a suo piacere il testo, e alla fine troverà il documento già formattato.

NOTA: risulta chiaro che avere tre Thread distinti(memorizzazione, manipolazione e formattazione) è più conveniente di avere tre Processi, in quanto i primi possono direttamente lavorare sulla stessa memoria mentre i secondi avrebbero memorie distinte.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

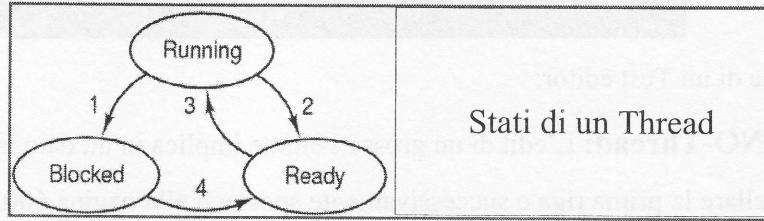
80

Offerte Thread

- 1) Rendono possibile mantenere l'idea di processo sequenziale che contiene chiamate di sistema bloccanti (input/output) ed avere comunque parallelismo.
- 2) Il blocco di un thread può essere evitato chiamando in precedenza una funzione di sistema che controlla se la chiamata di sistema presente all'interno del thread si bloccherà.
- 3) le chiamate di sistema bloccanti rendono più facile la programmazione e aumentano le prestazioni.
- 4) Condivisione delle risorse del Processo tra Thread.
- 5) Possibilità di bloccare solo un thread e non l'intero Processo.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

81



Simili a quelli dei processi e con le stesse caratteristiche.

Stack dei thread:

Lo Stack di ogni Thread contiene un record per ogni *procedura non conclusa*, le *variabile locali* della procedura e gli *indirizzi di ritorno* da usare quando la chiamata di procedura è terminata.

Esempio: Se all'interno di un Thread la procedura X chiama la procedura Y e questa chiama la procedura Z, allora le referenze delle procedure X,Y,Z risiedono tutte nello stack quando la procedura Z è in esecuzione.

78

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Uso dei Thread

Il motivo principale perché molte applicazioni fanno largo uso dei Thread è che in essi attività diverse procedono contemporaneamente ed alcune di esse possono bloccarsi di tanto in tanto.

Se tali applicazioni avessero un solo Thread, al verificarsi di un evento di blocco si interromperebbe l'intero processo.

Questo è uno dei motivi per cui, in applicazioni decomposte con moduli indipendenti, è auspicabile adottare i MultiThread.

Apparentemente stesso motivo dei processi paralleli....**MA NON DIMENTICHIAMO:**

- I Thread condividono lo spazio di indirizzamento e le risorse del Processo, questo introduce un nuovo elemento: la capacità di condividere, tra thread, lo spazio di indirizzamento la qual cosa NON può essere fatta tra processi.
- I Thread possono essere creati e distrutti più facilmente dei processi, e quando si cambia un thread dalla CPU NON si cambia il suo contesto (ciò non avviene con i processi paralleli).
- I Thread portano un guadagno in termini di prestazioni se assolvono a compiti diversi, migliorando le performance.

79

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Vantaggi Thread

- 1) Tempo di Risposta: trasformare un'applicazione interattiva in multithread può permettere ad un programma di continuare la sua esecuzione, anche in presenza di chiamate di sistema bloccanti.

Esempio:

Thread 1 - Gestione dell'interazione dell'interfaccia utente nel caso di consultazione WEB.

Thread 2 - caricamento dell'immagine dalla rete.

- 2) Condivisione delle risorse: tutti i Thread del processo sono presenti nello stesso spazio di indirizzamento.
- 3) Economia: assegnare memorie e risorse ad un processo, gestire i cambiamenti del contesto sono operazioni più costose che creare un thread.
- 4) Uso nel caso di multiprocessore: Un processo mono-thread in un sistema multiprocessore sarà eseguito soltanto su di uno di essi indipendentemente dal numero di processori presenti nel sistema. Lo stesso processo costituito da multithread consente la distribuzione dei thread sui vari processori, eseguendolo in modalità parallela e quindi migliorando le sue performance.

82

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread

Ci sono tre modi per implementare i Thread:

- nello spazio utente
- nel Kernel
- una implementazione mista.

83

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

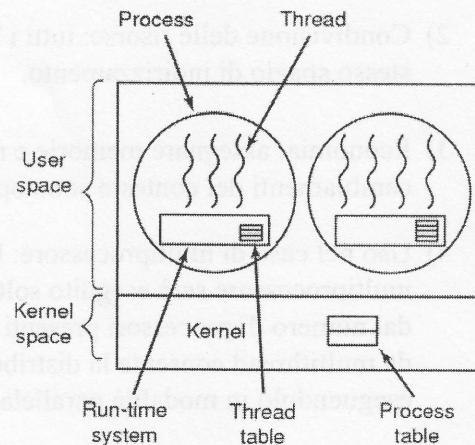
Implementazione dei Thread nello Spazio Utente

In tale implementazione i Thread sono realizzati nello spazio Utente, in questo modo il kernel non sa nulla di loro e gestirà i processi come un solo THREAD.

Il primo ovvio **Vantaggio**, per i thread definiti nello spazio utente, è che esso può essere implementato sui S.O. che non supportano Thread.

Definizione: Definiamo Run-Time System (Sistema a tempo di esecuzione) una collezione di procedure che gestiscono i Thread.

I Thread a livello utente sono gestiti dal Run-Time System.



<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

84

Implementazione dei Thread nello Spazio Utente

Quando i Thread sono gestiti nello spazio utente ogni processo ha bisogno della propria TABELLA dei THREAD per tenere traccia della loro esecuzione.

Tale tabella è analoga a quella dei processi e contiene informazioni sullo stack pointer, il program counter, i registri....

La tabella è gestita dal run-time system, esso si comporta come il Kernel del sistema operativo quando un processo passa da uno stato ad un altro.

Quando il Thread fa qualcosa per essere bloccato esso fa una **chiamata al run-time system** il quale si preoccupa di raccogliere tutte le informazioni del thread chiamante nella tabella dei thread, e attiva un nuovo thread.

Compiere **scambi di thread** in questo modo è estremamente rapido rispetto a fare una "trap" del kernel.....si pensi che cambiare un processo implica cambiare un contesto.

Inoltre con i thread a livello utente si possono definire propri **algoritmi di schedulazione**.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

85

Implementazione dei Thread nello Spazio Utente

Svantaggi dei Thread definiti dall'utente:

Implementazione alle chiamate bloccanti di sistema : inaccettabilità che i thread possano bloccare altri Thread, obiettivo fondamentale dei thread.

ES. Un Thread di lettura da tastiera mandato in esecuzione prima che il dato sia inserito.

Soluzione: definire una funzioni che testi in anticipo se una chiamata si bloccherà e agire di conseguenza.

Fault di pagina: una pagina non presente in memoria deve essere caricata dal disco, se il thread definito dall'utente per il caricamento della pagina non è riconosciuto dal Kernel, esso attenderà (attesa attiva) sino alla terminazione della lettura dal disco delle nuove pagine; altri thread del processo potrebbero essere pronti per la loro esecuzione.

Eccessivo uso dei Thread utente: in molti casi l'utente definisce i thread quando non è necessario, pesanti calcoli o gestione WEB in cui intervengono costantemente trap al kernel.

86

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread nello Spazio Utente

- La creazione dei Thread avviene con una chiamata al sistema che effettua la creazione o la cancellazione del Thread aggiornando la tabella.
- Quando i Thread sono gestiti dal Kernel, non occorre avere una tabella dei thread per ogni processo, poiché il kernel ha una visione generale del sistema, esso avrà una tabella comune di thread la quale tiene traccia di tutti i thread.
- Le informazioni presenti nella tabella dei thread del kernel sono analoghe a quelle definite nello spazio utente, ma la loro allocazione è nel kernel piuttosto che nello spazio utente.
- Le informazioni dei thread dei processi utente non sono distribuite nelle varie aree utente bensì sono raggruppate in un'unica area comune gestita dal Kernel.
- La tabella dei thread si affianca alla tabella dei processi per tenere traccia di essi.

87

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread

nel KERNEL

Svantaggi 1: la gestione di un thread nel KERNEL implica un maggior tempo di overhead in quanto ogni attività sui thread fa uso di una chiamata di sistema che risulta essere più onerosa di una semplice chiamata di procedura nel caso di thread definiti dall'utente.

Svantaggi 2: quando un Thread si blocca, il kernel decide quale thread presente nella tabella dei thread attivare. In generale tale thread può non appartenere allo stesso processo.

Cambio di contesto.

Ciò non avviene con i thread definiti nello spazio utente in cui il possesso presente nella CPU è di proprietà ancora dell'utente, e quindi con possibilità di esecuzioni di altri thread utente, se un thread utente ha finito la sua corsa.

NOTA: Con i thread definiti nel kernel l'utente rischia di avere ridotto il proprio tempo di possesso della CPU. (es. un thread si blocca prima del time-out, e il possesso passa ad un altro thread non necessariamente dello stesso processo in quanto il kernel pesca il thread dalla tabella dei thread comuni).

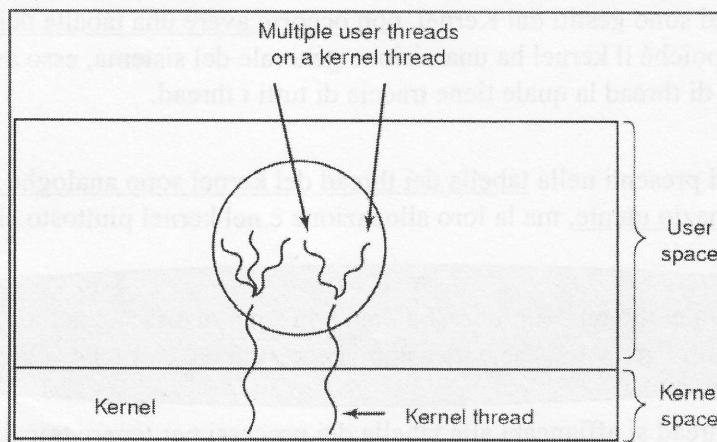
88

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread

in modo misto

Un modo ragionevole per acquisire i vantaggi dei Thread definiti a livello utente e le garanzie della modalità a livello Kernel è quella di definire i thread misti ossia:



Raggruppare più thread utenti in uno o più thread di kernel. In questo modo in generale si diminuisce il tempo di accesso alla CPU da parte del processo e si garantisce una più comoda gestione.

89

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread Attivazione dello Schedulatore

Attivare uno schedulatore nello spazio utente, per garantirsi da una parte buone prestazioni dovute allo spazio utente e dall'altra l'efficienza della gestione dei thread da parte del kernel.

Partendo dai thread definiti nello spazio utente:

L'Obbiettivo è di eliminare le transizioni non necessarie tra Spazio Utente e Spazio Kernel.

La strada da percorrere tende a eliminare sia chiamate di sistema non bloccanti sia il controllo anticipato del bloccaggio di un thread.

L'efficienza è ottenuta sfruttando al meglio il modulo run-time proprio dell'utente: quando un thread si blocca non occorre coinvolgere il kernel (si nasconde il blocco del Thread), il run-time, in presenza di tale evento, attiverà un thread proprio del processo, risparmiando così l'overhead della chiamata al sistema.

90

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Implementazione dei Thread Attivazione dello Schedulatore

Metodo: Il Kernel, rendendosi conto che un Thread è bloccato, effettua una chiamata al Run-Time (UPCALL) passandogli i riferimenti di tale thread, il Run-Time, prende atto di ciò, mette un flag nella specifica locazione nella tabella dei thread e attiva un nuovo thread appartenente al processo. Quando il vecchio Thread riparte il kernel lo comunica al Run-time dell'utente che lo rimette in gioco togliendo il precedente flag.

Nota: I kernel che è ad un livello più basso attiva un modulo di livello più alto (run-time) questo in generale va contro la filosofia dei S.O. a strati in cui il livello "n" offre servizi al livello "n+1".

91

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Comunicazione tra Processi

La comunicazione tra processi (InterProcess Communication) si snoda attraverso tre interrogativi:

- 1) come un processo può passare delle informazioni ad un altro?
- 2) come evitare che uno o più processi non si mettano l'uno sulla strada dell'altro, ossia evitare che un processo insegua un altro che è in una situazione critica?
- 3) come eseguire i processi in modo adeguato, quando esiste una dipendenza tra di essi?

Nota: i tre precedenti interrogativi sono da considerarsi validi anche per i Thread a parte

il primo la cui soluzione è nella struttura stessa dei thread (i thread condividono lo spazio degli indirizzi, e quindi anche i dati).

Le implicazioni seguenti saranno riferiti ai processi ma possono intendersi anche per i Thread.

92

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

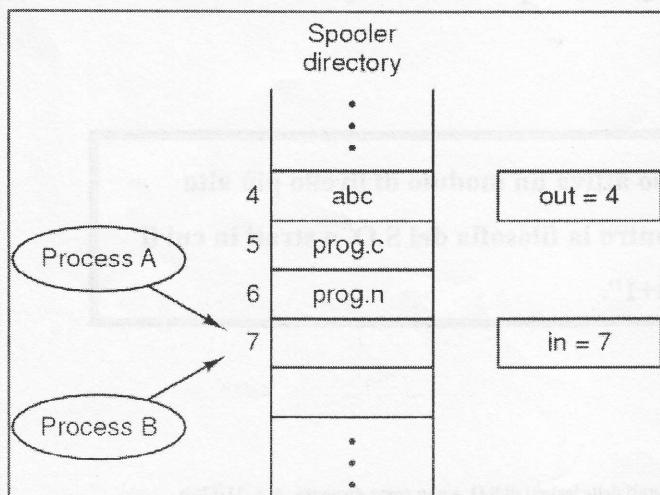
Comunicazione tra Processi

Corse critiche e un loro esempio

La situazione in cui due o più processi intendono leggere o scrivere un qualche dato su una zona condivisa e il risultato finale dipende dall'ordine con cui vengono eseguiti i processi viene detta una corsa critica (poiché l'ordine dei processi e quindi le relative azioni non può essere definito a priori).

Lo spooler di stampa:

Un processo, che necessita di stampare un file, aggiunge il nome del file da stampare in una directory speciale della "directory di spool"; un altro processo chiamato "demone della stampante" controlla periodicamente tale directory per effettuare le relative stampe e rimuove i loro nomi dopo aver effettuato la stampa.



Supponiamo la seguente condizione nella directory di stampa, i file nominati da 0 a 3 sono stati stampati quindi il file da stampare è il 4 ed il suo valore è residente nella variabile globale **OUT**, inoltre il primo posto libero in cui può essere inserito un nuovo file da stampare è il 7 il cui valore è definito nella variabile globale **IN**.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

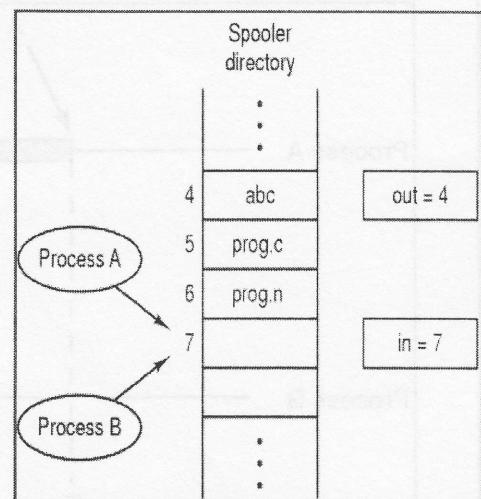
93

Condizione Critica

Consideriamo due processi A,B che devono stampare ciascuno un file sulla stessa stampante:

Step 1: Il processo A fa richiesta di stampa e legge dalla variabile globale IN il valore 7 che memorizza in una sua variabile locale.

Step 2: Il S.O. assegna la CPU al processo B in quanto è scaduto il quantum di tempo del processo A, in questo caso il processo B legge anch'esso il valore 7 nella variabile IN e aggiorna sia l'indirizzo 7 con il nome del file da stampare sia la variabile IN con 8. (Nota: a questo punto il processo B è sicuro che il suo file sarà stampato, ma non è così)



Step 3: il processo A riprende possesso della CPU, trova nella sua variabile locale l'indirizzo 7, scrive in tale posizione il nome del suo file, sovrascrivendo il nome del file del processo B che ancora deve essere stampato, aggiorna la variabile globale IN con 8 (incremento della variabile locale)....con queste condizioni il file del processo B non sarà stampato.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Comunicazione tra Processi Soluzioni alle Corse Critiche

Il Problema di fondo è cercare di evitare che più di un processo scriva su o legga da risorse condivise (risorse).

La soluzione è la tecnica della mutua esclusione, ossia se un processo sta usando una risorsa bisogna che gli altri siano bloccati all'uso della stessa risorsa.

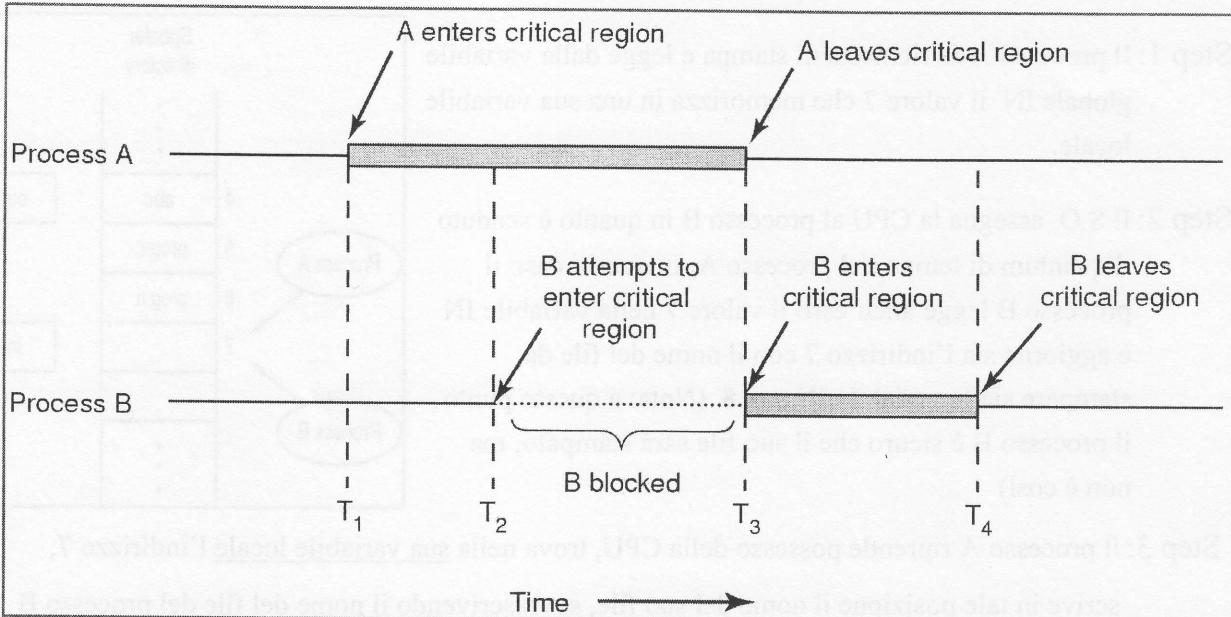
Definizione: Definiamo **Regione Critica** la parte del programma che può interferire con altri processi nell'uso di risorse comuni.

Per avere una buona soluzione al problema devono essere soddisfatte le seguenti condizioni:

- 1) Due processi non devono mai trovarsi contemporaneamente all'interno delle loro regioni critiche.
- 2) Non si deve fare alcuna ipotesi sulle velocità e sul numero delle CPU.
- 3) Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi.
(Un processo nella sua regine critica blocca ogni altro processo che tenta di entrare nella regine critica.)
- 4) Nessun processo deve aspettare indefinitivamente per poter entrare in una sezione critica.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

ESEMPIO



<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

96

La Mutua Esclusione con Attesa Attiva

Esaminiamo alcune metodologie per ottenere la mutua esclusione:

"Quando un processo si trova all'interno della sua regione critica nessun altro processo potrà entrare nella propria regione critica".

- Disabilitazione delle interruzioni;
- Variabili di Lock;
- Alternanza Stretta;
- La soluzione di Peterson;

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

97

Soluzione Semplice, Efficace e poco Attraente:

Permettere ad ogni processo di disabilitare le interruzioni non appena entra nella sua regione critica, e di riabilitarli non appena ne esce.

Questo approccio, in generale, non è molto attraente. Non è opportuno dare ad un processo utente il potere di disabilitare le interruzioni

La disabilitazione delle interruzioni inibisce l'uso della CPU da parte di altri utenti e del Kernel, con gravi crisi sul buon funzionamento del sistema.

Questo tipo di strategia può essere adottata dal kernel in quanto a volte il kernel, che si trova all'interno di una regione critica (aggiornamento liste, gestione processi), deve tenere bloccati tutti i processi per evitare corse critiche.

Conclusione

La strategia che disabilita gli interrupt è una tecnica utile nel kernel ma NON è opportuna per i processi utente.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Variabili Lock

La tecnica della variabile di lock è basata sulla *condivisione di una variabile booleana* che se trovata a 0 garantisce che nessun processo si trova all'interno della regione Critica.

Un processo prima di entrare nella propria regione critica controlla la variabile comune (variabile di lock) :

Se tale variabile è settata a 1 allora aspetta sino a quando un altro processo la settnerà a 0;

se il valore della variabile è 0 allora il processo assegna 1 alla variabile ed entrerà nella propria "regione critica",

terminata la sezione "regione critica" il processo settnerà a 0 il valore della variabile di lock, permettendo ad altri processi di entrare nelle proprie regioni critiche.

Conclusione

Questo approccio non è efficiente in quanto si possono verificare gli stessi effetti che si sono verificati nella directory di Spool delle stampanti.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

```
While (true)
{
```

```
    while (turno != 0);  

        regione_critica();  

        turno = 1;  

        regione_non_critica();
}
```

Processo Zero (0)

```
-----[turno=0]-----
```

```
While (true)
{
```

```
    while (turno != 1);  

        regione_critica();  

        turno = 0;  

        regione_non_critica();
}
```

Processo Uno (1)

- Il processo Zero legge la variabile turno, la trova a 0 ed entra nella sua regione_critica;
- Il processo Uno legge la variabile turno, la trova a 0 ed entra nel ciclo in cui testa la variabile turno; (*Attesa Attiva*: Attesa perché il processo Uno è fermo dentro un loop infinito, Attiva perché fa uso della CPU soltanto per testare la variabile di lock “turno”).
- Quando il processo Zero esce dalla sua regione_critica pone a 1 il valore di turno, permettendo al processo Uno di entrare nella sua regione_critica.

-----[Conclusione]-----

la turnazione dei processi non è una tecnica ottimale quando si è in presenza di processi in cui i tempi di elaborazione della regione NON critica sono estremamente diversi.

100

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

```
While (true)
{
```

```
    while (turno != 0);  

        regione_critica();  

        turno = 1;  

        regione_non_critica();
}
```

Esempio di Alternanza Stretta

```
-----[turno=0]-----
```

```
-----[turno=1]-----
```

```
While (true)
{
```

```
    while (turno != 1);  

        regione_critica();  

        turno = 0;  

        regione_non_critica();
}
```

P0

Quando il processo Zero lascia la regione_critica, mette turno a 1, entra nella propria regione NON critica e concede al processo Uno di entrare nella propria regione_critica.

Supponiamo che il processo Uno termini velocemente la sua regione_critica, mette a 0 turno e permetta al processo Zero di ESEGUIRE rapidamente la propria regione critica mettendo a 1 turno;

A questo punto i due processi si trovano nella regione non critica con turno settato a 1;

Poiché il tempo di elaborazione della regione non critica di P0 è estremamente veloce, essa termina e si pone per l'esecuzione della sezione critica, ciò non può avvenire perché la variabile turno è settata a 1, ossia P0 è bloccato da P1 quando questo si trova nella sua regione NON Critica (P0 deve attendere parte dell'esecuzione della sezione non critica di P1 e tutta la sua regione critica).

Violazione: P0 è bloccato da P1 quando questo non è nella sua regione critica, violazione della terza regola.

101

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Combinazione mista: variabili di lock e turni.

NOTA: La soluzione di Peterson è una soluzione corretta quando i processi hanno tutti la stessa priorità di esecuzione, inoltre tale soluzione implica un uso eccessivo della CPU.

```
#define FALSE = 0
#define TRUE = 1
#define N 2
int turno
int interessato[2];

void entra_nella_regione(int processo);
{
    int altri;
    altri = 1 - processo;
    interessato[processo] = TRUE;
    turno = processo;
    while (turno == processo && interessato[altri] == TRUE) ;
}

void lascia_la_regione(int processo)
{
    interessato[processo] = FALSE;
}
```

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

102

Sospensione e Risveglio

Consideriamo il caso in cui due processi sono a **priorità diverse** e osserviamo che la soluzione di Peterson mal si adatta a tale situazione.

ESEMPIO: Siamo dati due processi **H** e **L**, **H** a priorità alta (lo scheduler manda in esecuzione **H** non appena si trova nello stato di pronto) e **L** a priorità bassa (lo scheduler manda in esecuzione **L** quando questo si trova nello stato di pronto e nessun altro processo fa richiesta di CPU), ad un certo istante, mentre **L** è nella sua regione critica, **H** si trova nello stato di pronto quindi comincia la sua ATTESA ATTIVA (ciclo infinito aspettando che **L** esca dalla sua regione critica), poiché **L** è a bassa priorità non avrà mai il possesso della CPU quindi **H** (proprietario della CPU) non uscirà mai dall'ATTESA ATTIVA.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

103

Produttore / Consumatore

Due Processi P1 e P2 condividono un buffer a dimensione fissa e limitata, P1(produttore) deposita nel buffer informazioni mentre P2(consumatore) li preleva.

I problemi nascono quando:

- P1 tenta di depositare dati sul buffer pieno. La soluzione è di sospendere P1 per risvegiliarlo quando P2 avrà rimosso uno o più elementi.

- P2 tenta di prelevare dati da un buffer vuoto. La soluzione è di sospendere P2 per risvegiliarlo dopo che P1 avrà depositato uno o più elementi.

```
#define N 100           /* number of slots in the buffer */
int count = 0;          /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep(); /* repeat forever */
        /* if buffer is full, go to sleep */
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer); /* put item in buffer */
        /* increment count of items in buffer */
        /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep(); /* repeat forever */
        /* if buffer is empty, got to sleep */
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer); /* take item out of buffer */
        /* decrement count of items in buffer */
        consume_item(item); /* was buffer full? */
        /* print item */
    }
}
```

104

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Produttore / Consumatore

Considerazioni sul problema del produttore consumatore:

- Step1:** supponiamo di avere il buffer vuoto (count =0).
- Step2:** il consumatore legge count, lo trova a 0 e in quel momento lo scheduler attiva il produttore.
- Step3:** il produttore incrementa count, che assumerà valore 1 e manda una sveglia al consumatore che andrà persa.
- Step4:** lo scheduler attiva il consumatore che ha la variabile count = 0 attiva sleep.
- Step5: il produttore riempirà il buffer SENZA svegliare il consumatore.**

```
#define N 100           /* number of slots in the buffer */
int count = 0;          /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep(); /* repeat forever */
        /* if buffer is full, go to sleep */
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer); /* put item in buffer */
        /* increment count of items in buffer */
        /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep(); /* repeat forever */
        /* if buffer is empty, got to sleep */
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer); /* take item out of buffer */
        /* decrement count of items in buffer */
        consume_item(item); /* was buffer full? */
        /* print item */
    }
}
```

NOTA: L'essenza del problema è che una sveglia spedita ad un processo che non è sospeso, va persa, se non andasse persa andrebbe tutto bene.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

105

I semafori assolvono a tale compito: ***conservare il numero delle sveglie pervenute.***

Quindi un semaforo può avere valore 0, il che indica che non è stata salvata alcuna sveglia, o un numero positivo che indica che una o più sveglie sono state inviate.

Le operazioni di Sleep e Wakeup alla luce dei semafori sono definite come segue:

SLEEP (down) - la Sleep su di un semaforo controlla se il valore è maggiore di zero, se è tale allora decrementa il semaforo (consuma una delle sveglie che erano state salvate) e continua il suo compito, se il valore è 0 il processo viene sospeso e la sleep non viene completata.

WAKEUP (up) - L'operazione incrementa il valore del semaforo su cui viene invocata.

NOTA: Le operazioni Sleep e Wakeup sono **operazione atomiche**, ossia le operazioni di controllo del valore, cambio e eventuale sospensione sono indivisibili; attivata una istruzione di sleep o wakeup essa non può essere interrotta da nessun processo durante la sua esecuzione.

106

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Prodotto/Consumatore con i Semafori

Questa soluzione usa tre semafori:

1) EMPTY - quanti elementi del buffer sono vuoti, posto al numero massimo degli elementi. (*necessario per bloccare il Produttore*)

2) MUTEX - per garantire che il consumatore e il produttore non accedono contemporaneamente al buffer, posto a 1.

3) FULL - numero degli elementi del buffer che sono occupati, posto a 0. (*necessario per bloccare il Consumatore*)

In questo esempio si usano i semafori in modo differente:

Il semaforo MUTEX è usato per la mutua esclusione;

I semafori FULL e EMPTY sono necessari per la sincronizzazione, essi garantiscono che le sequenze di eventi si verifichino o no in un certo ordine.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

107

Altro metodo per la comunicazione tra processi è lo scambio dei messaggi tra di essi (Message Passing).

Tale metodo usa due primitive SEND e RECEIVE, che sono chiamate di sistema piuttosto che costrutti del linguaggio.

SEND (destinazione, &messaggio);

RECEIVE(sorgente, &messaggio);

La prima Procedura spedisce un messaggio al destinatario.

La seconda lo riceve da un mittente.

NOTA: se non è disponibile nessun messaggio il ricevente potrebbe bloccarsi sinché non ne arriva uno o in alternativa uscire con un controllo d'errore.

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

108

Problematiche inerenti lo Scambio dei Messaggi

Nota: Una delle problematiche inerente la comunicazione tra processi, realizzata con l'invio dei messaggi, si presenta quando i due processi che comunicano sono in esecuzione su due macchine distinte connesse tra di loro attraverso una rete dati.

In questo caso i messaggi possono essere persi per innumerevoli motivi.

Per evitare la perdita del messaggio: il mittente e il destinatario si mettono d'accordo utilizzando uno speciale messaggio di ricezione (conferma o **ACKNOWLEDGEMENT**). Se il mittente non riceve il messaggio entro un intervallo di tempo, il messaggio viene ritrasmesso.

Per evitare la perdita dell'acknowledgement: avviene quando il destinatario ha ricevuto il messaggio, e avendo inviato la conferma questa va persa. Il mittente non ricevendo la conferma rimanderà il messaggio che arriva al destinatario: duplicazione del messaggio.

Per evitare la duplicazione del messaggio: il mittente inserisce un numero di sequenza che il destinatario confronterà con i numeri di messaggio già pervenuti, in tal senso il destinatario potrà discriminare tra messaggi già ricevuti (duplicazione di numero sequenziale) e messaggi vecchi (numero sequenziale non registrato).

Evitare l'ambiguità del processo mittente e destinatario: ?

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

109

Produttore Consumatore con i Messaggi

Risoluzione del problema Produttore/Consumatore con lo scambio dei messaggi, senza uso della memoria condivisa.

In tale strategia il primo a fare la mossa **NONè il produttore**, così come si penserebbe logicamente, bensì il consumatore, questo manderà ad esempio 100 messaggi vuoti al produttore, che li acquisirà inserendogli le informazioni e rispedendoli indietro.

Se il Produttore lavorerà più velocemente del consumatore riempirà i 100 messaggi vuoti e aspetterà che il consumatore gli invii qualche messaggio vuoto, IL PRODUTTARE è BLOCCATO.

Se il Consumatore lavora più velocemente i 100 messaggi saranno vuoti e il CONSUMATORE è BLOCCATO.

I messaggi vengono affiancati da dei contenitori detti MAILBOX, abbastanza grandi da contenere N messaggi,

Se la mailbox è piena la SEND del produttore viene bloccata aspettando che il consumatore prelievi uno o più messaggi.

Se la mailbox è vuota la RECEIVE del consumatore viene bloccata aspettando che il produttore inserisca uno o più messaggi.

La bufferizzazione può essere eliminata inserendo la **Sincronizzazione** tra processi, Send del produttore e Receive del consumatore si bloccano aspettando rispettivamente la Receive o la Send.

110

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

Produttore Consumatore con i Messaggi

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                     /* message buffer */

    while (TRUE) {
        item = produce_item();                     /* generate something to put in buffer */
        receive(consumer, &m);                     /* wait for an empty to arrive */
        build_message(&m, item);                  /* construct a message to send */
        send(consumer, &m);                       /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                  /* get message containing item */
        item = extract_item(&m);                /* extract item from message */
        send(producer, &m);                     /* send back empty reply */
        consume_item(item);                   /* do something with the item */
    }
}
```

<<Questi handout sono da intendersi come appunti delle lezioni di S.O. e non come dispensa. A.A. 11/12>>

111