**Teoria Sistemi Operativi**

Descrivere il processo nel sistema Unix: caratteristiche, rappresentazione e principali system call per la sua gestione

In Unix il processo è l’**unità di computazione**. Esso viene definito **processo pesante con codice rientrante**: i dati non sono condivisi tra processi (modello ad ambiente locale), il codice può tuttavia essere condiviso.

Un processo, durante la sua esistenza può trovarsi in vari **stati**:

* **Init**, caricamento in memoria del processo e inizializzazione delle strutture dati del SO
* **Ready**, processo pronto ad acquisire la CPU
* **Running**, il processo usa la CPU
* **Sleeping**, il processo è sospeso in attesa di un evento
* **Terminated**, eliminazione del processo dalla memoria e dal SO
* **Zombie**, il processo è terminato, ma è in attesa che il processo padre ne rilevi lo stato di terminazione
* **Swapped**, il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria

Il descrittore del processo (**Process Control Block**) in Unix è rappresentato da 2 strutture dati distinte:

* **Process structure**, contenente le informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato in cui questo si trova)
* **User structure**, contenente le informazioni necessarie al sistema solo quando il processo risiede in memoria centrale (non è swapped)

Per quanto riguarda l’**immagine del processo** (insieme delle aree di memoria e strutture dati associate al processo) in Unix, essa non è totalmente accessibile in modo user: **parte** di **utente** (*e.g. area dati globali di utente*) e **parte** di **kernel** (*e.g. stack del kernel*). Inoltre non tutta l’immagine può essere sottoposta a trasferimento temporaneo in memoria secondaria: **parte** **swappable** (*e.g. user structure*) e **parte** **non swappable** (*e.g. process structure*).

Le system call di Unix sono attivabili attraverso chiamate a funzioni di librerie C standard. Le seguenti sono le principali per la gestione dei processi:

* int **fork**(), consente a un processo di generare un processo figlio. Padre e figlio condividono lo stesso codice, il figlio eredita una copia dei dati (di   
  utente e di kernel) del padre. La funzione restituisce un intero che:
* per il processo creato (figlio) vale 0
* per il processo padre è un valore positivo   
  che rappresenta il PID (Process Identifier) del processo figlio
* è un valore negativo in caso di errore

—> Il diverso valore restituito consente di discriminare il codice eseguito dal padre e quello eseguito dal figlio.

* void **exit**(int status), permette a un processo di terminare volontariamente. Prevede un parametro (status) mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione (ad esempio esito dell’esecuzione). È sempre una chiamata senza ritorno.
* int **wait**(int\* status), consente al processo che la invoca di rilevare lo stato di terminazione dei processi figli. Il parametro status è l’indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio. Il risultato prodotto dalla wait() è il pid del processo terminato, oppure un codice di errore (<0).
* int **exec**(«i parametri dipendono dal tipo di exec»), vengono sostituiti codice ed eventuali argomenti di invocazione del processo che la chiama con codice e argomenti di un programma specificato come parametro. È una chiamata senza ritorno, se ritorna significa che la chiamata è fallita, pertanto il processo continua ad eseguire il codice iniziale. Esistono diverse varianti nella famiglia exec: execl(), execlp(), execve(), …

Descrivere lo scheduling della CPU

In un SO multiprogrammato è di fondamentale importanza garantire il massimo utilizzo della CPU. Per tale motivo e per garantire ad ogni processo l’acquisizione della risorsa, una parte del SO è dedicata alla realizzazione del cosiddetto scheduling della CPU. Essa prende il nome di **scheduler a breve termine** (short-term scheduler).

Lo scheduler decide a quale processo assegnare la CPU tra quelli presenti nella coda dei processi pronti (ready queue); a seguito della decisione, viene attuato il cambio di contesto (context switch).

Il **dispatcher** è la parte di SO che realizza il cambio di contesto.

I processi nella ready queue che dovranno essere eseguiti vengono scelti in base ad un algoritmo di scheduling. Esistono 2 principali categorie di algoritmi di scheduling:

* **senza prelazione** (non pre-emptive), la CPU rimane allocata al processo running finché esso non si sospende volontariamente o non termina
* **con prelazione** (pre-emptive): il processo running può essere prelazionato, cioè il SO può sottrargli la CPU per assegnarla ad un nuovo processo. I sistemi a divisione di tempo hanno sempre uno scheduling pre-emptive

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni indicatori di performance:

* **Utilizzo della CPU**: percentuale media di utilizzo della CPU nell’unità di tempo (da massimizzare)
* **Throughput** (del sistema): numero di processi completati nell’unità di tempo
* **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue
* **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento
* **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l’inizio della prima risposta (a differenza del turnaround, non dipende dalla velocità dei dispositivi di I/O)

Algoritmi di scheduling principali:

* **First Come First Served**, la coda dei processi pronti viene gestita in modo FIFO: i processi sono schedulati secondo l’ordine di arrivo nella coda. L’algoritmo non è pre-emptive
* **Shortest Job First**, per ogni processo nella ready queue, viene stimata la lunghezza del prossimo CPU-burst (fase in cui viene impiegata soltanto la CPU senza interruzioni dovute a operazioni di I/O). Viene schedulato il processo con il CPU-burst più corto. L’algoritmo può essere pre-emptive o meno, in caso affermativo prende il nome di **Shortest Remaining Time First** (se nella coda arriva un processo con CPU-burst minore del CPU-burst rimasto al processo —> pre-emption)
* **Round Robin**, algoritmo pre-emptive tipicamente usato in sistemi time sharing, la ready queue è gestita in modo FIFO. Ad ogni processo viene allocata la CPU per un intervallo di tempo costante Δt (quanto di tempo). Il processo usa la CPU per Δt oppure si blocca prima. Allo scadere del quanto di tempo avviene la revoca della CPU e il re-inserimento del processo nella coda dei processi pronti.
* **Multiple Level Feedback Queues**, sono presenti più code, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, …). Ogni coda ha una diversa priorità e viene gestita con un algoritmo di scheduling distinto, eventualmente diverso (es. FCFS o Round Robin).

I processi possono muoversi da una coda all’altra, in base alla loro storia:

* passaggio da priorità bassa ad alta <— processi in attesa da molto tempo (feedback positivo)
* passaggio da priorità alta a bassa <— processi che hanno già utilizzato molto tempo di CPU (feedback negativo)

Descrivere la sincronizzazione tra processi tramite segnali in Unix: meccanismi e principali system call

La sincronizzazione permette di imporre vincoli sull'ordine di esecuzione delle operazioni dei processi interagenti.

Nel modello ad ambiente globale la sincronizzazione tra processi concorrenti è realizzata mediante strumenti di sincronizzazione software (*e.g. semafori*). Nel modello ad ambiente locale essa è invece realizzata tramite scambio di eventi.

Unix adotta il **modello ad ambiente locale** e la sincronizzazione può realizzarsi mediante **scambio di segnali**.

Un segnale è un’interruzione software, che notifica un evento in modo asincrono al processo che la riceve. Il processo destinatario rileva l’evento istantaneamente, qualunque sia il suo stato.

Un segnale può essere inviato:

* dal kernel a processi utente
* da un processo utente ad altri processi utente (*e.g. system call* ***kill()***)

Quando un processo riceve un segnale, può comportarsi in tre modi diversi (a meno che non si tratti dei segnali SIGKILL o SIGSTOP, in quel caso è possibile solo l’azione di default):

* gestire il segnale con una funzione **handler** definita dal programmatore
* eseguire un’azione predefinita dal SO (**azione di default**)
* ignorare il segnale (**nessuna reazione**)

Nei primi due casi, il processo reagisce in modo asincrono al segnale:

1. interruzione dell’esecuzione
2. esecuzione dell’azione associata (handler o default)
3. ritorno alla prossima istruzione del codice del processo interrotto

Per ogni versione di Unix esistono vari tipi di segnale, ognuno identificato da un intero. Ogni segnale è associato a un particolare evento e prevede una specifica azione di default.

Principali system call per la sincronizzazione tramite segnali:

* void(\***signal**(int sig, void (\*func)(int)))(int), il processo che la invoca sceglie di gestire esplicitamente il segnale sig (intero identificativo del segnale), scegliendo l’azione associata ad esso tramite il campo func. func può essere un puntatore all’handler o valere SIG\_IGN o SIG\_DFL, rispettivamente segnale ignorato e azione di default. Ritorna un puntatore al precedente gestore del segnale, o SIG\_ERR in caso di errore
* int **kill**(int pid, int sig), può essere usata da un processo per inviare un segnale sig (intero identificativo del segnale) ad un altro processo pid (PID del processo destinatario).
* unsigned int **sleep**(unsigned int N), provoca la sospensione del processo per N secondi (al massimo). Se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato prematuramente. Restituisce il numero di secondi non utilizzati dell’intervallo di sospensione
* unsigned int **alarm**(unsigned int N), imposta un timer che dopo N secondi invierà al processo il segnale SIGALRM. Restituisce 0 se non vi erano time-out impostati in precedenza, altrimenti il numero di secondi mancante allo scadere del time-out precedente
* int **pause**(), sospende il processo fino alla ricezione di un qualunque segnale

Descrivere le strutture dati e i meccanismi per l’accesso ai file in Unix

In Unix l’accesso ai file avviene in modo **sequenziale**, esso viene visto come una sequenza di byte. Il puntatore al file (**I/O Pointer**) registra la posizione corrente.

Sono disponibili vari modi di accesso: lettura, scrittura, lettura/scrittura, …

L’accesso è subordinato all’operazione di apertura: per poter essere acceduto il file deve prima essere aperto.

Strutture dati per l’accesso a file:

* **tabella dei file aperti di processo**: si trova nella user structure di ciascun processo e contiene un elemento per ogni file aperto dal processo. L’indice di ogni elemento della tabella viene detto **file descriptor**. Ha dimensione limitata (tipicamente 20 elementi). I file descriptor 0, 1, 2 individuano rispettivamente standard input, output, error (aperti automaticamente)
* **tabella dei file aperti di sistema**: si trova nell’area dati del kernel e contiene un elemento per ogni sessione di accesso a file nel sistema (a processi diversi che accedono allo stesso file, corrispondono entry distinte). Ogni entry contiene l’I/O pointer (che indica la posizione corrente all’interno del file) e un puntatore all’i-node del file nella tabella dei file attivi.
* **tabella dei file attivi**: si trova nell’area dati del kernel e contiene un elemento per ogni file aperto nel sistema. Ogni entry contiene una copia del suo i-node, in questo modo si rendono più efficienti le operazioni di accesso, evitando accessi al disco per ottenere attributi dei file acceduti

L’apertura di un file provoca:

1. l’inserimento di un elemento (individuato da un file descriptor) nella prima posizione libera della tabella dei file aperti del processo
2. l’inserimento di un nuovo record nella tabella dei file aperti di sistema
3. l’eventuale copia dell’i-node nella tabella dei file attivi (se il file non è già in uso in qualche altro processo)

La chiusura di un file provoca:

1. l’eliminazione della relativa entry dalla tabella dei file aperti del processo
2. l’eventuale eliminazione dell’elemento corrispondente dalla tabella dei file aperti di sistema (se il file non è condiviso con altri processi)
3. l’eventuale eliminazione dell’elemento corrispondente dalla tabella dei file attivi (se il file non è aperto da altri processi)

L’accesso al file è fatto tramite file descriptor. Ogni operazione di lettura/scrittura (per la quale è sempre garantita singolarmente l’atomicità) agisce sequenzialmente sul file, a partire dalla posizione corrente del I/O pointer.

Descrivere le tecniche di allocazione della memoria secondaria

Le tecniche più comuni per l’allocazione dei blocchi sul disco sono:

* **allocazione contigua**, ogni file è mappato su un insieme di blocchi fisicamente contigui.

Vantaggi:

* basso costo per la ricerca di un blocco
* possibilità di accesso sequenziale e diretto

Svantaggi:

* individuazione dello spazio libero per l’allocazione di un nuovo file
* frammentazione esterna: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili —> necessità di azioni di compattazione
* aumento dinamico delle dimensioni di file
* **allocazione a lista concatenata**, i blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata.

Vantaggi:

* non c’è frammentazione esterna
* minor costo di allocazione

Svantaggi:

* possibilità di errore se link danneggiato
* maggior occupazione dello spazio fisico (spazio occupato dai puntatori)
* difficoltà di realizzazione dell’accesso diretto
* alto costo della ricerca di un blocco
* **allocazione a indice**, A ogni file è associato un blocco indice in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file.

I vantaggi sono gli stessi dell’allocazione a lista, più:

* possibilità di accesso diretto
* maggiore velocità di accesso (rispetto a liste)

Svantaggi:

* scarso utilizzo dei blocchi indice per file di piccole dimensioni

Esistono SO che adottano più di un metodo di allocazione, spesso: allocazione contigua per file piccoli, allocazione a indice per file grandi.

Unix adotta l’allocazione ad indice a più livelli di indirizzamento (il blocco indice può contenere non solo riferimenti a blocchi dati, ma anche riferimenti ad altri blocchi indice).

Descrivere l’organizzazione fisica del file system Unix

Nel file system Unix vi è **omogeneità**: tutto è file.

Esistono 3 categorie di file:

* **file ordinari**
* **direttori**
* **dispositivi fisici**: file speciali

Ad ogni file possono essere associati uno o più nomi simbolici ma ad ogni file è associato uno ed un solo descrittore (**i-node**), univocamente identificato da un intero (**i-number**). Tra gli attributi contenuti nell’i-node vi si trova: tipo di file, proprietario e gruppo, dimensione, bit di protezione, …

Il metodo di allocazione utilizzato in Unix è ad **indice a più livelli di indirizzamento** (l’essere a più livelli di indirizzamento implica che il blocco indice può contenere non solo riferimenti a blocchi dati, ma anche riferimenti ad altri blocchi indice. Tecnica necessaria per l’allocazione di file di grandi dimensioni).

La superficie del disco File System è partizionata in 4 regioni:

* **boot block**, contiene le procedure di inizializzazione del sistema
* **super block**, fornisce: i limiti delle 4 regioni, il puntatore a una lista dei blocchi liberi e il puntatore a una lista degli i-node liberi
* **i-list**, è una lista contenente tutti i descrittori (i-node) dei file ordinari, direttori e dispositivi fisici presenti nel file system. L’accesso ad ogni elemento avviene tramite indice (i-number)
* **data blocks**, è l’area del disco effettivamente disponibile per la memorizzazione dei file; contiene i blocchi allocati e i blocchi liberi

Descrivere le tecniche di allocazione della memoria centrale

Per allocare codice e dati dei processi in memoria centrale esistono varie tecniche:

* **Allocazione contigua**:
* **Partizione singola**, la parte di memoria disponibile per l’allocazione dei processi di utente non è partizionata, ergo u‍n solo processo alla volta può essere allocato in memoria.

Svantaggio: assenza di multiprogrammazione.

* **Partizioni multiple a dimensione fissa**, la memoria fisica disponibile per l’allocazione dei processi è suddivisa a priori in un numero prefissato di partizioni. Quando un processo viene caricato, il SO cerca una partizione libera di dimensione sufficiente ad accogliere il suo spazio di indirizzamento.

Svantaggi: frammentazione interna; grado di multiprogrammazione limitato al numero di partizioni; dimensione massima dello spazio di indirizzamento di un processo limitata dalla dimensione della partizione più estesa.

* **Partizioni multiple a dimensione variabile**, ogni partizione è creata dinamicamente e dimensionata in base alla dimensione del processo da allocare: quando un processo viene caricato, SO cerca un’area della memoria fisica sufficientemente grande per allocarvi dinamicamente la partizione associata.

Vantaggi rispetto alla tecnica a partizioni fisse: assenza di frammentazione interna, grado di multiprogrammazione variabile, dimensione massima dello spazio di indirizzamento di un processo non limitata.

Svantaggi: scelta dell’area in cui allocare (varie politiche possibili); frammentazione esterna —> necessità di compattazione periodica.

* **Allocazione non contigua**:
* **Paginazione**, si effettua un partizionamento dello spazio fisico di memoria in pagine fisiche (frame) di dimensione DIM costante, sulle quali è possibile mappare porzioni dei processi da allocare (pagine logiche, sempre di dimensione DIM costante).

Vantaggi: pagine logiche contigue possono essere allocate su pagine fisiche non contigue —> non c’è frammentazione esterna; la frammentazione interna è limitata dalla dimensione del frame; è possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo se si fa uso di memoria virtuale.

* **Segmentazione**, è un’estensione della tecnica di allocazione a partizioni variabili. Lo spazio logico degli indirizzi di un processo è suddiviso in parti (segmenti), ognuna caratterizzata da nome e lunghezza propri. I segmenti sono divisi per semantica (codice, dati, heap, …) e ognuno di essi è allocato in memoria in modo contiguo. Ad ogni segmento è inoltre possibile applicare diritti di accesso specifici. Il SO associa ad ogni segmento un intero identificativo.

Descrivere la paginazione

La paginazione è una tecnica di **allocazione non contigua** di codice e dati dei processi in memoria centrale.

Essa consiste nel partizionamento dello spazio fisico di memoria in **pagine fisiche** (**frame**) di dimensione costante e limitata, sulle quali mappare porzioni dei processi da allocare.

Lo spazio fisico viene visto come un insieme di frame di dimensione DIM. Lo spazio logico è invece suddiviso in un insieme di **pagine logiche** di dimensione sempre uguale a DIM. Ogni pagina logica di un processo viene mappata su una pagina fisica in memoria centrale.

Struttura dell’indirizzo logico:

< p , d >

dove p è il numero della pagina logica, mentre d è l’offset della cella rispetto all’inizio della pagina.

Struttura dell’indirizzo fisico:

< f , d >

dove f è il numero del frame, mentre d è l’offset della cella rispetto all’inizio del frame.

Il binding tra indirizzi logici e fisici può essere realizzato mediante una struttura dati, detta **tabella delle pagine**. Ogni tabella delle pagine è associata ad un particolare processo: ogni elemento significativo della tabella associa ad una pagina logica p, la pagina fisica f nella quale p è allocata, l’offset rimane invariato.

Essa può essere molto grande e la traduzione degli indirizzi deve essere il più veloce possibile. Per superare questi problemi la realizzazione di tale struttura dati può essere fatta:

* su registri di CPU, accesso veloce ma cambio di contesto pesante e dimensioni limitate della tabella
* in memoria centrale, registro **Page Table Base Register** memorizza la collocazione della tabella in memoria
* in memoria centrale + cache, **Translation Look-aside Buffers** per velocizzare l’accesso

La tabella delle pagine ha dimensione fissa e non sempre viene utilizzata completamente. Per distinguere gli elementi significativi da quelli non utilizzati ogni elemento della table contiene un **bit di validità:** se è a 1 la entry è valida (pagina appartiene allo spazio logico del processo), se invece è 0 no. Inoltre esiste il **Page Table Length Register**, un registro contenente il numero degli elementi validi nella tabella delle pagine.

Vantaggi della paginazione:

* Pagine logiche contigue possono essere allocate su pagine fisiche non contigue —> non c’è frammentazione esterna.
* Le pagine sono di dimensione limitata —> la frammentazione interna è limitata dalla dimensione del frame
* È possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo —> supporto alla realizzazione di memoria virtuale

Descrivere la segmentazione

La segmentazione è una tecnica di allocazione della memoria centrale che si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti (**segmenti**), ognuna caratterizzata da nome e lunghezza propri. Ogni segmento è diviso per semantica (*e.g. codice, dati, stack, heap*) ed è allocato in memoria in modo contiguo. A ciascun segmento è possibile applicare diritti di accesso specifici (*e.g. codice sola lettura*), inoltre gil SO associa un intero attraverso il quale lo si può riferire.

Struttura dell’indirizzo logico:

< s , d >

dove s è l’ identificatore numerico che individua il segmento nel sistema, mentre d è l’offset della cella rispetto all’inizio del segmento.

Il binding tra indirizzi logici e fisici può essere realizzato mediante una struttura dati, detta **tabella dei segmenti**. Ogni tabella dei segmenti è associata ad un particolare processo. Ha una entry per ogni segmento, che ne descrive l’allocazione in memoria fisica mediante la coppia < base, limite >, dove base indica l’indirizzo della prima cella del segmento nello spazio fisico, limite indica invece la dimensione del segmento.

La tabella dei segmenti può essere realizzata:

* su registri di CPU, accesso veloce ma cambio di contesto pesante e dimensioni limitate della tabella
* in memoria centrale, mediante registro **Segment Table Base Register**, il quale memorizza la collocazione della tabella in memoria e registro **Segment Table Length Register** contenente il numero degli elementi validi nella tabella
* In memoria centrale + cache

La segmentazione può essere vista come un’estensione della tecnica di allocazione a partizioni variabili. Infatti quest’ultima prevede un’unica partizione per processo, mentre la prima più partizioni (segmenti) per processo. Per questo motivo, come per la tecnica a partizioni variabili, rimane il problema della **frammentazione esterna**. Esso può tuttavia essere superato tramite allocazione dei segmenti con tecniche ad hoc (best fit, worst fit, …) e compattazione periodica.

Descrivere la memoria virtuale

La dimensione della memoria può rappresentare un vincolo importante, riguardo a

* **dimensione dei processi**
* **grado di multiprogrammazione**

Può essere desiderabile un sistema di gestione della memoria che:

* consenta la presenza di più processi in memoria, indipendentemente dalla dimensione dello spazio disponibile
* svincoli il grado di multiprogrammazione dalla dimensione effettiva della memoria

La memoria virtuale è un metodo di gestione della memoria che consente l’**esecuzione di processi non completamente allocati in memoria**, in modo completamente trasparente per il programmatore e per i processi stessi.

Vantaggi:

* dimensione spazio logico degli indirizzi non vincolata dall'estensione della memoria
* grado di multiprogrammazione indipendente dalla dimensione della memoria fisica
* efficienza: caricamento di un processo e swapping hanno un costo più limitato (meno I/O)
* astrazione: il programmatore non deve preoccuparsi dei vincoli relativi alla dimensione della memoria

Di solito la memoria virtuale è realizzata mediante tecniche di **paginazione su richiesta**: tutte le pagine di ogni processo risiedono in memoria secondaria; durante l’esecuzione alcune di esse vengono trasferite, all’occorrenza, dal pager (un modulo ad hoc del SO) dalla memoria secondaria alla memoria centrale, e viceversa.

Descrivere il trattamento del page fault nei sistemi operativi con memoria virtuale

L’utilizzo della memoria virtuale in un SO prevede che una pagina dello spazio logico di un processo può essere allocata in memoria centrale o trovarsi in memoria secondaria.

Per distinguere i due casi ogni elemento della **tabella delle pagine** contiene 1 **bit di validità**, il cui valore indica:

* 1: la pagina è presente in memoria centrale
* 0: la pagina è in memoria secondaria oppure è invalida, cioè non fa parte dello spazio logico del processo

Nella traduzione di ogni indirizzo logico si consulta la tabella delle pagine; se il bit di validità della pagina richiesta è 0 viene inviata un interruzione al SO (page fault).

Quando il kernel del SO riceve l’interruzione dovuta al page fault:

1. Salvataggio del contesto di esecuzione del processo (registri, stato, tabella delle pagine)
2. Verifica del motivo del page fault (mediante una tabella interna al kernel). Se si tratta di un riferimento illegale (violazione delle politiche di protezione) il SO termina il processo. Se invece il riferimento è legale la pagina è in memoria secondar
3. Copia della pagina in un frame libero
4. Aggiornamento della tabella delle pagine
5. Ripristino del contesto del processo ed esecuzione dell’istruzione interrotta dal page fault

Descrivere la sovrallocazione e gli algoritmi di sostituzione nei sistemi operativi con memoria virtuale

In seguito a un page fault, se è necessario caricare una pagina in memoria centrale, può darsi che non ci siano frame liberi —> è necessaria la sovrallocazione, cioè la sostituzione di una pagina “vittima” PV allocata in memoria con la pagina “nuova” PV da caricare. I passaggi effettuati sono i seguenti:

1. Individuazione della vittima PV
2. Salvataggio di PV su disco
3. Caricamento di PN nel frame liberato
4. Aggiornamento tabella delle pagine
5. Ripresa del processo

Per rendere più efficiente il trattamento del page fault in caso di sovrallocazione si introduce in ogni elemento della tabella delle pagine un bit di modifica (**dirty bit**). Se è a 1 la pagina è stata aggiornata da quando è caricata in memoria, se è a 0 invece non è stata modificata. L’algoritmo di sostituzione può quindi esaminare il bit di modifica della vittima nella tabella delle pagine e eseguire swap-out della pagina solo se il dirty bit è settato.

Esistono diversi algoritmi di sostituzione, la cui finalità è sostituire quelle pagine la cui probabilità di essere accedute a breve termine è bassa. I principali sono:

* **Least Frequently Used**, viene sostituita la pagina che è stata usata meno frequentemente in un intervallo di tempo prefissato. Per implementarlo è necessario associare un contatore degli accessi ad ogni pagina —> la vittima è quella con minimo valore del contatore
* **First In First Out**, si sostituisce la pagina che è da più tempo caricata in memoria centrale, indipendentemente dal suo uso. Per realizzarlo è necessario memorizzare la cronologia dei caricamenti in memoria: time stamping o gestione di una coda
* **Least Recently Used**, viene sostituita la pagina che è stata usata meno recentemente. Per implementarlo va registrarata la sequenza degli accessi alle pagine in memoria: time stamping o gestione di uno stack. Di solito è il più preferibile per il rispetto del principio di località.

Descrivere i passaggi di sostituzione ed elaborazione nella shell Bash

All’esecuzione di un file comandi contenente istruzioni eseguibili dalla shell Bash, vengono effettuati i seguenti passaggi in ordine:

* preparazione e collegamento dei comandi come filtri: **ridirezione e piping** di ingresso/uscita
* **sostituzione dei comandi**, i comandi contenuti tra ` ` (backquote) sono eseguiti e sostituiti dall’output prodotto
* **sostituzione delle variabili e dei parametri**, i nomi delle variabili ($nome) sono espansi nei valori corrispondenti
* **sostituzione dei metacaratteri**, i metacaratteri \* ? [ ] sono espansi nei nomi di file secondo un meccanismo di pattern matching

Ogni passaggio può essere saltato se non c’è motivo che venga fatto (*e.g. nello script non sono presenti comandi tra backquote*).

L’espansione non è effettuata se i caratteri speciali sono inibiti da alcuni caratteri con un significato speciale, questi sono:

* \ (backslash), il carattere successivo è considerato come un carattere normale
* ‘ ‘ (apici singoli), i caratteri al loro interno non vengono espansi
* “ ” (doppi apici), i caratteri al loro interno non vengono espansi, con l’eccezione di $ \ ` `

Esporre possibili soluzioni al problema della mutua esclusione

La disponibilità di risorse comuni nel modello ad ambiente globale, pone il problema dell’accesso esclusivo da parte dei processi concorrenti. La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo. Nessun vincolo è imposto sull’ordine con il quale le operazioni sulle variabili vengono eseguite.

Una corretta soluzione al problema della mutua esclusione deve soddisfare i seguenti requisiti:

1. Sezioni critiche della stessa classe devono essere eseguite in modo mutuamente esclusivo
2. Quando un processo si trova all’esterno di una sezione critica non può impedire l’esecuzione della stessa sezione (o di sezioni della stessa classe) ad altri processi
3. Assenza di deadlock

Lo schema generale per la risoluzione del problema della mutua esclusione prevede che ogni processo, prima di entrare in una sezione critica, esegua una serie di istruzioni che gli garantiscano l’accesso esclusivo alla risorsa, se questa è libera, oppure ne impediscano l’accesso se questa è già occupata (**prologo**); se la risorsa è occupata il prologo deve provocare l’attesa del processo che lo esegue. Al completamento dell’azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la risorsa (**epilogo**).

Soluzioni possibili al problema della mutua esclusione:

* **soluzioni algoritmiche**, non richiedono particolari meccanismi di sincronizzazione ma sfruttano semplicemente la possibilità di condivisione di variabili (*e.g. algoritmo di Dekker o algoritmo di Peterson*)
* **soluzioni hardware-based**, il supporto è fornito direttamente dall’architettura hardware (*e.g. disabilitazione delle interruzioni, lock/unlock*)
* **soluzioni basate su strumenti di sincronizzazione**, prologo ed epilogo sfruttano strumenti software per la sincronizzazione realizzati dal nucleo del sistema operativo che consentono l’effettiva sospensione dei processi in attesa di eseguire sezioni critiche (*e.g. semafori*)

Descrivere le variabili condizione: definizione, uso e possibili semantiche dell’operazione signal

Le variabili condizioni sono usate all’interno del monitor.

Una variabile condizione rappresenta una **coda** (**condition queue**) nella quale i processi possono sospendersi se la **condizione di sincronizzazione** non è verificata (condizione logica che assicura un certo ordinamento nell’accesso alla risorsa controllata dal monitor).

E’ possibile applicare ad ogni variabile condizione (cond) due operazioni:

* **wait(cond)**; la cui esecuzione sospende il processo, introducendolo nella coda individuata dalla variabile cond, e il monitor viene liberato. Al risveglio, il processo riacquisisce l’accesso mutamente esclusivo al monitor e riprende l’esecuzione.
* **signal(cond)**; la cui esecuzione riattiva un processo in attesa nella coda individuata dalla variabile cond; se non vi sono processi in coda, non produce effetti.

Quando un processo effettua una signal(), entrambi i processi (quello “segnalante” Q e quello “segnalato” P), possono proseguire la loro esecuzione. Il monitor tuttavia, per definizione, limita a 1 il numero di processi che possono eseguire al suo interno. A chi va data la precedenza?

Possibili semantiche dell’operazione signal():

**signal\_and\_wait**: P riprende immediatamente l’esecuzione e Q viene sospeso nella **entry queue** (coda relativa alla mutua esclusione) —> precedenza al processo segnalato.

**signal\_and\_continue**: Q prosegue la sua esecuzione mantenendo l’accesso esclusivo al monitor, dopo aver riattivato P, il quale si sospende nella entry queue —> precedenza al processo segnalante.

La semantica signal\_and\_continue implica che la condizione di sincronizzazione relativa alla variabile condition su cui è stata fatta la signal, possa non essere più verificata al momento dell’accesso al monitor di P. Infatti poiché altri processi possono entrare nel monitor prima di P, questi potrebbero modificare la condizione di sincronizzazione (lo stesso potrebbe fare Q).   
E’ pertanto necessario che, al rientro nel monitor, il processo segnalato P verifichi di nuovo la condizione di sincronizzazione —> test della condizione con while anziché if.

Descrivere e confrontare la gestione a controllo di programma e la gestione a interruzioni di dispositivi di I/O

Nella gestione di ogni dispositivo di I/O, la sincronizzazione tra CPU e periferica può avvenire secondo 2 modelli alternativi:

* gestione a controllo di programma (o polling)
* gestione basata su interruzioni

Nella gestione a controllo di programma il processo applicativo prepara un comando e lo invia al dispositivo esterno depositandolo nel **registro di controllo** del suo **controllore**. Contestualmente setta il **bit di start** (bit di start = 1). A questo punto attende che il dispositivo esterno termini l’esecuzione del comando (viene controllato il bit di flag nel **registro di stato**). Quando il **bit di flag** viene settato (bit di flag = 1) il processo può infine prelevare l’esito dell’esecuzione dal registro di stato, ed eventualmente i dati dal **registro dati** se presenti. Se il processo interno deve inviare altri comandi alla periferica ripete il ciclo.

Il problema principale del polling è il ciclo di attesa attiva con cui il processo applicativo controlla se la periferica ha terminato l’esecuzione del comando e reso disponibile esito e dati —> la CPU viene utilizzata inutilmente. Lo schema a controllo di programma, in particolare, non è adatto a sistemi multiprogrammati.

Per evitare l’attesa attiva è possibile usare la gestione a interruzioni. Essa prevede di riservare un semaforo per ogni dispositivo e di settare il **bit di abilitazione alle interruzioni** (nel registro di controllo del controllore della periferica interessata) (bit di abilitazione alle interruzioni = 1). Così facendo il processo che attende la terminazione del comando da parte del dispositivo esterno, si sospende nella coda individuata dal semaforo anziché impegnare la CPU. Quando il dispositivo ha terminato invia una interrupt e il processo si può infine risvegliare.