Università Ca' Foscari Venezia - Corso di Laurea in Informatica
Sistemi Operativi – prof. Augusto Celentano

Domande di riepilogo

1) Il descrittore di processo (*Process Control Block*) include al suo interno:

(a) l'identificatore del processo e quello del processo padre;
(b) l'identificatore del processo ma non quello del processo padre;
(c) l'identità dell'evento di cui il processo è in attesa;
(d) una copia dello stato del processore al momento dell'ultima *preemption* o sospensione del processo;
(e) le aree dati, codice e stack del processo;
(f) la sola area stack del processo, utilizzando puntatori alle page table per il resto;
(g) i puntatori alle page table che descrivono la memoria virtuale assegnata al processo;
(h) le variabili condivise con altri processi nelle sezioni critiche;
(i) nulla di quanto elencato sopra.

2) Quali delle seguenti affermazioni sul modello di processo a sette stati sono corrette?

(a) è consentita la transizione Running –> Blocked/Suspend;
(b) è consentita la transizione Running –> Running/Suspend;
(c) è consentita la transizione Blocked/Suspend –> Ready;
(d) è consentita la transizione Blocked/Suspend –> Running;
(e) è consentita la transizione Blocked/Suspend –> Ready/Suspend;
(f) è consentita la transizione Blocked/Suspend –> Exit;
(g) è consentita la transizione New –> Ready/Suspend;
(h) è consentita la transizione New –> Blocked/Suspend;
(i) nessuna delle affermazioni precedenti è corretta.

3) In un sistema operativo time sharing senza priorità, un processo in esecuzione può essere interrotto dallo scheduler e portato nello stato di pronto prima dello scadere del quanto di tempo?

(a) no, perché altrimenti il processo verrebbe rallentato e potrebbe essere soggetto a *starvation*
(b) no, perché lo scopo di un sistema time sharing, in assenza di priorità, è quello di eseguire tutti i processi con gli stessi diritti (*fair scheduling*)
(c) no, perché in un sistema time-sharing senza priorità lo scheduler non interrompe mai un processo in esecuzione, ma lascia che il processo si sospenda da solo
(d) sì, perché altrimenti gli altri programmi non potrebbero continuare la loro esecuzione
(e) sì, se il processo esegue sempre le stesse istruzioni, perché potrebbe essere in un loop infinito

4) La strategia di scheduling Shortest Process Next (SPN) privilegia i processi:

(a) CPU bound, perché essendo a bassa priorità utilizzano poco la CPU
(b) I/O bound, perché hanno sempre una priorità maggiore dei processi CPU bound
(c) I/O bound, perché un processo che si sospende spesso per operazioni di I/O utilizza la CPU per un tempo breve ad ogni burst
(d) che hanno iniziato l'esecuzione da poco tempo, perché si pensa che dureranno complessivamente meno dei processi più vecchi
(e) che usano poca memoria, perché la strategia SPN ottimizza i processi che usano poche risorse

5) Si consideri lo scheduling della CPU in un sistema operativo multiprogrammato. Quale di queste politiche di scheduling può rinviare in modo indefinito l'esecuzione di qualche processo, causando quindi il fenomeno detto starvation?

(a) Shortest Process Next, perché privilegia sempre i processi con CPU burst brevi contro i processi con CPU burst lunghi, che quindi possono aspettare un tempo indefinito
(b) FCFS, perché eseguendo sempre i primi processi in coda si possono ritardare all'infinito i processi entrati in coda successivamente
(c) Round Robin con quanto di tempo sufficiente lungo, perché se ci sono molti processi nella coda di round-robin gli ultimi possono attendere un tempo indefinito
(d) Real-time con priorità variabili, perché può alzare la priorità dei processi urgenti e ridurre la priorità dei processi meno prioritari, ritardando in modo indefinito la loro esecuzione

6) Una strategia di scheduling si dice *pre-emptive* quando:

(a) interviene per scegliere un processo pronto da eseguire solo quando il processo in esecuzione si sospende per un'operazione di I/O
(b) può interrompere il processo in esecuzione anche se questo non si sospende da solo, se un nuovo processo entra nello stato di pronto
(c) realizza una gestione di tipo *batch*
(d) favorisce i processi I/O bound
(e) favorisce i processi CPU bound

7) Un processo passa dallo stato di esecuzione allo stato di pronto

(a) quando fa una richiesta di I/O
(b) quando un processo con priorità più alta entra nello stato di pronto
(c) quando arriva un interrupt di clock
(d) quando un processo con priorità più bassa entra nello stato di pronto

8) In un sistema time sharing un processo può essere interrotto in modo forzato prima della scadenza del suo quanto di tempo

(a) no, mai
(b) sì, se ci sono altri processi in stato di attesa
(c) solo se c'è un processo pronto con priorità più alta
(d) solo se c'è un processo pronto con priorità più bassa

9) In un sistema real-time con priorità statiche un processo può restare in stato di pronto per un tempo indefinito

(a) no, perché il sistema operativo lo porterà in stato di esecuzione se l'attesa diventa troppo lunga
(b) sì, ma solo se nel sistema ci sono processi con priorità più alta

10) In un sistema time-sharing senza priorità ci sono questi processi:

P1 in esecuzione
P2 in attesa di un'operazione di I/O
P3 in stato di pronto

Quando si completa l'operazione di I/O, succede che:

(a) P1 rimane in esecuzione
(b) P1 va in stato di pronto
(c) P2 va in stato di pronto
(d) P2 va in stato di esecuzione al posto di P1
(e) P3 va in esecuzione al posto di P1

11) In un sistema operativo UNIX, le chiamate di sistema della famiglia exec()(execl(), execv(), etc. ...):

(a) sono gli unici meccanismi di creazione di nuovi processi;
(b) sono i principali meccanismi di creazione di nuovi processi;
(c) causano la terminazione del processo in corso e l'avvio di un nuovo processo;
(d) causano la sostituzione del processo in corso con uno nuovo;
(e) riportano come risultato il PID del nuovo processo;
(f) riportano come risultato il valore restituito dalla funziona main() dell'eseguibile specificato;
(g) nessuna delle affermazioni precedenti è corretta.

12) Nel sistema operativo Unix la funzione exec("nome_file", argomenti) ha queste proprietà:

(a) crea un nuovo processo che esegue il programma memorizzato in "nome_file";
(b) chiama una funzione del sistema operativo (system call) memorizzata in "nome_file"
(c) fa eseguire dal processo che la chiama un nuovo processo di nome "nome_file", cui passa come parametri gli argomenti elencati nella chiamata, e al termine riprende l'esecuzione del processo chiamante
(d) sostituisce il codice del processo che la chiama con il codice del programma memorizzato in "nome_file", e conserva i dati del processo originale
(e) sostituisce sia il codice sia l'area dati del processo che la chiama con il codice e l'area dati del programma memorizzato in "nome_file"

13) Nel sistema operativo Unix, la chiamata di sistema fork():

(a) sostituisce al processo esistente un nuovo processo, identico in tutto e per tutto al processo chiamante, tranne che nel Process Control Block (PCB);
(b) sostituisce al processo esistente un nuovo processo, costruito sulla base del file eseguibile passato come argomento alla chiamata, mantenendo il PCB del processo originale;
(c) divide il processo esistente in due processi che condividono il PCB: il primo identico al processo originale, il secondo costruito a partire dall'eseguibile passato come argomento alla chiamata;
(d) genera un nuovo processo, identico in tutto e per tutto al processo chiamante, tranne che nel Process Control Block (PCB);
(e) genera un nuovo processo, lanciando una nuova istanza del processo chiamante a partire dal file eseguibile corrispondente;
(f) è l'unico meccanismo disponibile con cui un processo può generare un altro processo;
(g) è il meccanismo più frequentemente usato dai processi per generare nuovi processi.

14) Dopo l'esecuzione di questo frammento di programma eseguito dal processo P

```
i = fork();
j = fork();
```

(a) ci sono due processi
(b) ci sono tre processi
(c) ci sono quattro processi

15) Si consideri il seguente codice C eseguito nel sistema operativo Unix:

```
1    i = fork();
2    printf (“1\n”);
3    if (i == 0) {
4      printf(“2\n”);
5      exec(“Programma_A”);
6    }
7    printf(“3\n”);
8    exec(“Programma_B”);
```

Alla riga 1 il processo (padre) crea un altro processo (figlio). Che cosa succede durante l’esecuzione, supponendo che non si verifichino errori?

Alla riga 2:

(a) Solo il processo padre scrive “1”
(b) Solo il processo figlio scrive “1”
(c) Sia il processo padre sia il processo figlio scrivono “1”

Alla riga 4:

(a) Solo il processo padre scrive “2”
(b) Solo il processo figlio scrive “2”
(c) Sia il processo padre sia il processo figlio scrivono “2”

Alla riga 7:

(a) Solo il processo padre scrive “3”
(b) Solo il processo figlio scrive “3”
(c) Sia il processo padre sia il processo figlio scrivono “3”

Alla riga 8:

(a) Il processo padre esegue il programma “Programma_B”
(b) Il processo figlio esegue il programma “Programma_B”
(c) Sia il processo padre sia il processo figlio eseguono il programma “Programma_B”

16) Un sistema di gestione di memoria virtuale che usa una politica di sostituzione delle pagine FIFO (First In First Out) in caso di necessità sceglie per la sostituzione

(a) la pagina che non è stata utilizzata da più tempo
(b) la pagina che non è stata modificata da più tempo
(c) la pagina che risiede in memoria da più tempo
(d) la pagina caricata in memoria per ultima
(e) la pagina con l’indirizzo fisico più basso

17) Con riferimento alla memoria virtuale a pagine e a segmenti, quali di queste affermazioni sono vere e quali false:

(a) in una memoria virtuale a segmenti i segmenti sono tutti della stessa lunghezza
(b) in una memoria virtuale a pagine le pagine possono essere di dimensioni diverse
(c) la memoria virtuale a segmenti e a pagine sono mutuamente esclusive
(d) nella memoria virtuale a pagine la corrispondenza tra indirizzo logico e indirizzo fisico è data da una formula
(e) nessuna delle affermazioi precedenti è vera

18) Con riferimento alla paginazione su richiesta (demand paging) quali di queste affermazioni sono vere e quali sono false:

(a) minimizza l'uso della memoria perché vengono caricate esclusivamente le pagine che servono per una specifica esecuzione
(b) permette ad un processo di essere eseguito in uno spazio fisico più piccolo dello spazio logico di indirizzamento utilizzato dal relativo programma
(c) se si richiede una pagina non presente in memoria il processo termina con una segnalazione di errore
(d) si deve sapere in anticipo quali pagine logiche saranno utilizzate dal processo
(e) richiede che si sappia in anticipo quante pagine fisiche saranno necessarie per completare l'esecuzione del processo

19) In un sistema a memoria virtuale, la relazione tra spazio logico e spazio fisico di indirizzamento è tale che

(a) lo spazio fisico è sempre maggiore dello spazio logico
(b) lo spazio fisico è sempre minore dello spazio logico
(c) lo spazio fisico di indirizzamento può essere indifferentemente maggiore o minore dello spazio logico

20) In un sistema a memoria paginata può verificarsi frammentazione esterna di memoria

(a) no
(b) sì
(c) sì, ma solo se la dimensione della pagina è grande

21) Un sistema di gestione di memoria virtuale che usa una politica di sostituzione delle pagine LRU in caso di necessità sceglie per la sostituzione

(a) la pagina che risiede in memoria da più tempo
(b) la pagina che non è stata utilizzata da più tempo
(c) la pagina che non è stata modificata da più tempo
(d) la pagina caricata in memoria per ultimo

Risposte

Sono riportate qui le risposte esatte di tutte le domande, con l'avvertenza che è importante ragionare sulle risposte stesse e darne una motivazione, anche perché una domanda sintetica può rivelare, ad un'analisi più attenta, alcuni sottintesi più o meno evidenti che si possono prestaere a interpretazioni articolate, portando a conclusioni diverse. Per questo motivo le risposte ad alcune domande (1, 8, 11, 14) sono motivate per esteso a titolo di esempio.

1) a, c, d, g

Gli elementi tipici di un PCB possono essere raggruppati in:

- identificatori del processo
- informazioni sullo stato del processore (al momento dell'ultimo prerilascio o sospensione)
- informazioni sul controllo del processo.

Segue direttamente che l'affermazione (d) è vera. Tra gli identificatori del processo abbiamo l'identificatore (ID) del processo, l'ID del processo parent e l'ID dell'utente. L'affermazione (a) è quindi corretta mentre la (b) è errata.

Tra le informazioni di controllo del processo un sottoinsieme di queste descrivono lo stato e la schedulazione: stato del processo (ad esempio in esecuzione, pronto o in attesa), priorità nella schedulazione, informazioni relative alla schedulazione dipendenti dall'algoritmo di scheduling ed infine l'identità dell'evento di cui il processo è in attesa prima di passare nello stato di pronto. L'affermazione (c) è quindi corretta.

Altre informazioni incluse tra le informazioni di controllo riguardano: privilegi del processo, risorse utilizzate (ad esempio i file aperti), gestione della memoria (include ad esempio puntatori a segmenti e/o page table della memoria virtuale del processo). L'affermazione (g) è quindi vera.

Tra le informazioni di controllo del processo possono essere inoltre presenti puntatori per collegare il processo in particolari strutture e flag/messaggi per l'interprocess communication. Il PCB non contiene informazioni in merito alle variabili condivise con altri procesi nelle sezioni critiche; l'affermazione (h) è falsa.

Il PCB, insieme al programma eseguito dal processo, i dati utenti e lo stack di sistema compone la cosiddetta immagine del processo.

Aree dati, codice e stack non sono quindi contenute nel PCB: sono false le affermazioni (e) ed (f), e per quanto detto sopra anche l'affermazione (i) è falsa.

2) e, g

3) c (la risposta b può essere considerata corretta, ma è una conseguenza e non uno scopo dell'algoritmo)

4) c

5) a (la risposta d è apparentemente plausibile, ma in realtà non lo è perché le priorità variabili sono utilizzare per compensare situazioni che possono portare a starvation e non per favorirle)

6) b

7) b (anche la risposta c è corretta se si ipotizza che all'interrupt di clock il processo corrente venga temporaneamente posto in stato di pronto prima di valutare se è necessario un context switch. Questa soluzione è più lineare ma meno efficiente.)

8) a oppure c, in funzione della assenza o presenza di priorità

In un sistema time sharing puro l'esecuzione viene ripartita a turno tra i processi della stessa priorità. In un sistema senza priorità è vera l'affermazione (a) e sono false le altre, ma un sistema senza priorità è una semplificazione didattica non realistica.

In un sistema con priorità i processi più prioritari hanno precedenza. Quindi se un processo a priorità *p* è in esecuzione e durante il suo quanto di tempo un processo di priorità $q > p$ entra nello stato di pronto (ad es. perché ha terminato un'attesa su un evento esterno) il processo in esecuzione viene interrotto. E' perciò vera l'affermazione (c).

9) b

10) a, c

11) g

Le risposte (a) e (b) non sono corrette in quanto nel sistema operativo UNIX la creazione di un processo avviene con la chiamata di sistema *fork().* La *fork()crea* una copia del processo chiamante (parent process). Restituisce al processo *parent* il PID del processo creato (child *process)* mentre restituisce 0 al processo *child.*

La chiamata ad una funzione della famiglia *exec* comporta la sostituzione dell'immagine del processo chiamante con l'eseguibile specificato dagli argomenti, lasciandone inalterato *pid* e processo *parent.* Il processo non viene quindi terminato: l'affermazione (c) è falsa.Inoltre, essendo sostituiti solo alcuni elementi del processo, e non il processo stesso, anche la risposta (d) è falsa.

Se hanno successo le funzioni della famiglia *exec* non effettuano ritorno al chiamante (il codice chiamante non esiste più, essendo sostituito dal nuovo eseguibile). In caso di errore, cioè se il nuovo eseguibile non ha potuto sostituire il codice del chiamante) viene restituito il valore -1. Quindi anche le risposte (e) ed (f) sono errate. Segue che solo la risposta (g) è vera.

12) e

13) d, f

14) c

Dopo l'esecuzione della prima istruzione i = fork () viene creato un secondo processo *P*'

che esegue lo stesso codice del processo *P.* Entrambi i processi proseguono con l'esecuzione dell'istruzione j = fork(), perciò entrambi creano un nuovo processo. *P* crea un processo *P*'' e *P*' crea un processo *P*'''*.* Se non si verificano errori, i processi alla fine dell'esecuzione del frammento di programma sono quattro.

15) riga 2: c; riga 4: b; riga 7: a; riga 8: a

16) c

17) e

18) sono vere solo le risposte a, b

19) c

20) a

21) b