

Sistemi Operativi, Proff. Mancini e Massaioli
Raccolta di esami ed esercizi svolti

A cura di:
Prof L.V. Mancini, Prof. F. Massaioli,
Dott. M. Conti, PhD, Dott. E. Sangineto, PhD

May 5, 2009

Introduzione

L'obiettivo di questa dispensa è fornire un supporto per la preparazione all'esame scritto di Sistemi Operativi dei Proff. Mancini e Massaioli. In queste pagine trovate degli esercizi di autovalutazione raggruppati per argomento. Alcuni di questi esercizi sono tipici di una prova scritta. Per trarre utilità dalla dispensa il consiglio è sforzarsi di risolvere da soli gli esercizi e consultare la soluzione proposta solo per verificare la propria comprensione. Si sottolinea che la dispensa *non sostituisce le lezioni in aula nè il libro di testo*. Il suo spirito è di essere uno strumento di verifica delle conoscenze acquisite. Inoltre, si sconsiglia fortemente di imparare "mnemonicamente" le risposte senza comprenderle: lo scopo di qualsiasi esame è proprio indagare il livello di comprensione dello studente.

Al fine di migliorare questa dispensa vi invitiamo ad indirizzare eventuali segnalazioni a Mauro Conti (conti@di.uniroma1.it) o Enver Sangineto (sangineto@di.uniroma1.it). Grazie e buono studio!

Per informazioni in merito alla struttura del compito ed alla modalità d'esame fare riferimento alla sezione "Modalità d'esame" nella pagina web del corso:

<http://twiki.di.uniroma1.it/twiki/view/Sistemioperativi1>.

Negli esercizi della prima parte del compito di esame è solitamente richiesta solo l'indicazione delle risposte corrette (risposta multipla); in questa dispensa per alcuni esercizi è stata fornita anche la motivazione delle risposte.

Contents

1	Processi	5
1.1	Testi	6
1.1.1	Chiamate di sistema della famiglia Exec	6
1.1.2	Chiamata di sistema Fork	6
1.1.3	Pseudocodice con Fork ed Exec	6
1.1.4	Descrittore di processo	7
1.1.5	Modello di processo a sette stati	7
1.1.6	Modello di processo a cinque stati	7
1.2	Soluzioni	7
1.2.1	Chiamate di sistema della famiglia Exec	7
1.2.2	Chiamata di sistema Fork	8
1.2.3	Pseudocodice con Fork ed Exec	8
1.2.4	Descrittore di processo	8
1.2.5	Modello di processo a sette stati	8
1.2.6	Modello di processo a cinque stati	8
2	Scheduling	9
2.1	Testi	10
2.1.1	Scheduler Round-Robin	10
2.1.2	Scheduler Highest Responce Ratio Next	10
2.1.3	Scheduler First Come First Served (FCFS)	10
2.1.4	Scheduling e starvation	11
2.1.5	Scheduling e starvation 2	11
2.1.6	Scheduler programmabile	11
2.1.7	Prerilascio di un processo	12
2.1.8	Esercizio scheduler Round-Robin	12
2.1.9	Esercizio scheduler SRT	12
2.1.10	Secondo esercizio di scheduler Round-Robin	12
2.1.11	Terzo esercizio di scheduler Round-Robin	13
2.1.12	Esercizio di scheduler Highest Response Ratio Next	13
2.2	Soluzioni	13
2.2.1	Scheduler Round-Robin	13
2.2.2	Scheduler Highest Responce Ratio Next	13
2.2.3	Scheduler First Come First Served (FCFS)	13
2.2.4	Scheduling e starvation	13
2.2.5	Scheduling e starvation 2	14
2.2.6	Scheduler programmabile	14
2.2.7	Prerilascio di un processo	15
2.2.8	Esercizio scheduler Round-Robin	15
2.2.9	Esercizio scheduler Shortest Remaining Time	15
2.2.10	Secondo esercizio scheduler Round-Robin	16
2.2.11	Terzo esercizio scheduler Round-Robin	16
2.2.12	Esercizio scheduler Highest Responce Ratio Next	17

3	Gestione della memoria	18
3.1	Testi	19
3.1.1	Translation Lookaside Buffer	19
3.1.2	Gestione della memoria con segmentazione	19
3.1.3	Peculiarità dei sistemi di gestione della memoria	19
3.1.4	Gestione della memoria, il fenomeno della frammentazione interna	19
3.1.5	Politiche di sostituzione delle pagine	20
3.1.6	Obiettivi nella gestione della memoria	20
3.1.7	Algoritmo del Clock	20
3.1.8	Sequenza di accessi in memoria con algoritmi LRU e Clock	20
3.1.9	Traduzione indirizzi con paginazione	20
3.1.10	Secondo esercizio su Traduzione indirizzi con paginazione	21
3.1.11	Traduzione degli indirizzi con segmentazione	21
3.2	Soluzioni	22
3.2.1	Translation Lookaside Buffer	22
3.2.2	Gestione della memoria con segmentazione	22
3.2.3	Peculiarità dei sistemi di gestione della memoria	22
3.2.4	Gestione della memoria, il fenomeno della frammentazione interna	22
3.2.5	Politiche di sostituzione delle pagine	22
3.2.6	Obiettivi nella gestione della memoria	22
3.2.7	Algoritmo del Clock	23
3.2.8	Sequenza di accessi in memoria con algoritmi LRU e Clock	23
3.2.9	Traduzione indirizzi con paginazione	23
3.2.10	Secondo esercizio su Traduzione indirizzi con paginazione	24
3.2.11	Traduzione degli indirizzi con segmentazione	24
4	Gestione I/O	25
4.1	Testi	26
4.1.1	Proprietà della tecnica del Buffering	26
4.1.2	DMA	26
4.1.3	Obiettivi del disk scheduling	26
4.1.4	Buffering	26
4.1.5	SSTF	26
4.1.6	SCAN	27
4.2	Soluzioni	27
4.2.1	Proprietà della tecnica del Buffering	27
4.2.2	DMA	27
4.2.3	Obiettivi del disk scheduling	28
4.2.4	Buffering	28
4.2.5	SSFT	28
4.2.6	SCAN	28
5	File System	30
5.1	Testi	31
5.1.1	Proprietà degli i-node	31
5.1.2	Struttura degli i-node	31
5.2	Soluzioni	31
5.2.1	Proprietà degli i-node	31
5.2.2	Struttura degli i-node	31
6	Concorrenza	32
6.1	Testi	33
6.1.1	Semafori	33
6.1.2	Mutua esclusione	33
6.1.3	Sezione critica	33
6.1.4	Problema lettori-scrittori	33
6.1.5	Comunicazione tra processi tramite scambio di messaggi	34
6.1.6	Race condition	34
6.1.7	Chiosco	34

6.1.8	Incrocio	35
6.1.9	Buffer	35
6.1.10	Implementazione di un semaforo generico e di un semaforo binario	35
6.1.11	Ponte	35
6.1.12	Filari	36
6.1.13	Filari con stallo	36
6.1.14	Fast Food	37
6.2	Soluzioni	37
6.2.1	Semafori	37
6.2.2	Mutua esclusione	37
6.2.3	Sezione critica	37
6.2.4	Problema lettori-scrittori	37
6.2.5	Comunicazione tra processi tramite scambio di messaggi	38
6.2.6	Race condition	38
6.2.7	Chiosco	38
6.2.8	Incrocio	41
6.2.9	Buffer	42
6.2.10	Implementazione di un semaforo generico e di un semaforo binario	43
6.2.11	Ponte	45
6.2.12	Filari	49
6.2.13	Filari con stallo	50
6.2.14	Fast Food	51
7	Stallo	54
7.1	Testi	55
7.1.1	Attesa indefinita	55
7.1.2	Prevenzione, esclusione e rimozione dell'attesa circolare	55
7.1.3	Condizioni dello stallo	55
7.2	Soluzioni	55
7.2.1	Attesa indefinita	55
7.2.2	Prevenzione, esclusione e rimozione dell'attesa circolare	55
7.2.3	Condizioni dello stallo	55

Chapter 1

Processi

1.1 Testi

1.1.1 Chiamate di sistema della famiglia Exec

(punti: -1,4)

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; No, il principale è la `fork()`
- (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 funzione `main()` dell'eseguibile specificato; ✗
- ✗ nessuna delle affermazioni precedenti è corretta. ✓

Non ritornano nulla al chiamante
Perché il chiamante non esiste più,
si è trasformato nel nuovo eseguibile

1.1.2 Chiamata di sistema Fork

(punti: -1,4)

In un 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; ✗ a e b sbagliate perché non sostituisce nulla ma genera un nuovo processo
- (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; ✗ non viene passato nessun eseguibile (quello è `exec`)
- ✓ (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.

1.1.3 Pseudocodice con Fork ed Exec

(punti: -1,4)

La seguente porzione di pseudocodice di un processo P_p contiene una chiamata alla primitiva `fork()` di UNIX che termina con successo e genera un processo figlio P_c . È possibile invece che la successiva invocazione della primitiva `exec()` fallisca. codice-A, eseguibile-B, codice-C, codice-D non possono causare errori di esecuzione e non contengono istruzioni di salto.

```
pid = fork();
```

```
if (pid > 0)
```

```
    codice-A;
```

```
else if (pid == 0)
```

```
    exec("eseguibile-B");
```

```
else
```

```
    codice-C;
```

```
    codice-D
```

Il codice D viene eseguito solo dal padre (quasi) -> se sono il figlio la mia var pid sarà 0 perché la `fork()` da 0 quando crea il figlio, di conseguenza, entra nell'`exec` e quindi non ritornerà mai al chiamante (guarda domanda `exec` di prima)

Se dovesse fallire la `exec()` ritorna -1 e quindi il figlio in questo caso esegue il codice-D

- (a) Il processo P_c esegue sicuramente il codice-A e potrebbe eseguire il codice-D;
- (b) il processo P_c esegue sicuramente il codice-A e non esegue mai il codice-D;
- (c) il processo P_p esegue sicuramente il codice-A e potrebbe eseguire il codice-D;
- (d) il processo P_p esegue sicuramente il codice-A e non esegue mai il codice-D;
- (e) il processo P_c esegue sicuramente l'eseguibile-B e potrebbe eseguire il codice-D;
- (f) il processo P_c esegue sicuramente l'eseguibile-B e non esegue mai il codice-D;
- (g) il processo P_c potrebbe eseguire il codice-C e potrebbe eseguire il codice-D;
- (h) il processo P_c esegue sicuramente il codice-C e non esegue mai il codice-D;
- (i) il processo P_p esegue sicuramente l'eseguibile-B e potrebbe eseguire il codice-D;

- (j) il processo P_p esegue sicuramente l'eseguibile-B e non esegue mai il codice-D;
- (k) il processo P_p potrebbe eseguire il codice-C e potrebbe eseguire il codice-D;
- (l) il processo P_p potrebbe eseguire il codice-C e non esegue mai il codice-D;
- ☒ (m) nessuna delle affermazioni precedenti è corretta.

1.1.4 Descrittore di processo

(punti: -1,4)

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'ultimo prerilascio 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.

1.1.5 Modello di processo a sette stati

(punti: -1,4)

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

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

1.1.6 Modello di processo a cinque stati

(punti: 9)

Illustrare in al più 100 parole il modello di processo a cinque stati.

1.2 Soluzioni

1.2.1 Chiamate di sistema della famiglia Exec

Risposte corrette: (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*). Torna al processo *parent* il PID del processo creato (*child process*) mentre torna 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 contrario viene tornato il valore -1. Quindi anche le risposte (e) ed (f) sono errate.

Da quanto detto segue che l'affermazione (g) è vera.

1.2.2 Chiamata di sistema Fork

Risposte corrette: (d), (f).

1.2.3 Pseudocodice con Fork ed Exec

Risposte corrette: (m).

1.2.4 Descrittore di processo

Risposte corrette: (a), (c), (d), (g).

Il *process control block* (PCB) è una struttura che, per ogni processo, contiene tutte le informazioni necessarie al sistema operativo. Gli elementi tipici di un PCB possono essere raggruppati in:

- identificatori del processo
- informazioni sullo stato del processore (al momento dell'ultimo pririlascio 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 è errata la (b).

Tra le informazioni di controllo del processo un sottinsieme 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 processi 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). Da quanto detto segue che l'affermazione (i) è falsa.

1.2.5 Modello di processo a sette stati

Risposte corrette: (e), (g).

1.2.6 Modello di processo a cinque stati

Il modello prevede gli stati: New, Ready, Running, Blocked, Exit. Le transizioni possibili sono le seguenti.

- Null→New: un nuovo processo è creato.
- New→Ready: il sistema operativo accetta il processo per l'esecuzione.
- Ready→Running (*dispatch*): il sistema operativo seleziona (tramite lo *scheduler* o *dispatcher*) uno dei processi pronti per l'esecuzione.
- Running→Exit: il processo termina o è abortito.
- Running→Ready: al processo viene sottratto il processore (*preemption*).
- Running→Blocked: un processo bloccato chiede qualcosa per cui è necessario attendere prima di continuare l'esecuzione.
- Blocked→Ready: si verifica un evento per cui era in attesa un processo bloccato

Chapter 2

Scheduling

2.1 Testi

2.1.1 Scheduler Round-Robin

(punti: -1,4)

In uno *scheduler Round-Robin*:

- (a) l'avanzamento relativo dei processi è sincronizzato e dipende solo dai tempi di arrivo, dato che tutti i processi ricevono ciclicamente a turno un quanto di tempo, nell'ordine in cui sono entrati nel sistema;
- ☒ (b) l'avanzamento relativo dei processi è di fatto asincrono e imprevedibile perché i processi possono fare richieste bloccanti di dati che arrivano asincronamente dall'esterno del sistema;
- ☒ (c) l'avanzamento relativo dei processi è di fatto asincrono e imprevedibile perché influenzato dalle interruzioni che il sistema riceve;
- (d) ogni processo ha un quanto di tempo di durata differente, che deve essere nota in anticipo;
- (e) ogni processo ha un quanto di tempo di durata differente, determinata con una media sulla base delle esecuzioni precedenti;
- ☒ (f) se la durata del quanto di tempo tende a infinito, il comportamento tende a quello di uno *scheduler FCFS* (*First Come First Served*);
- (g) se la durata del quanto di tempo tende a infinito, il comportamento tende a quello di uno *scheduler SPN* (*Shortest Process Next*).

2.1.2 ~~Scheduler Highest Response Ratio Next~~

(punti: -1,4)

Uno *scheduler Highest Response Ratio Next*:

- (a) non utilizza la *preemption*, ed attende che il processo in corso sia completato;
- (b) non utilizza la *preemption*, ed attende che il processo in corso sia completato o passi in stato di *blocked*;
- (c) forza la *preemption* del processo in esecuzione all'arrivo nel sistema di un processo con rapporto R maggiore;
- (d) forza la *preemption* del processo in esecuzione non appena il suo rapporto R diventa inferiore a quello di uno dei processi in stato di *ready*;
- (e) causa la *starvation* dei processi lunghi;
- (f) causa la *starvation* dei processi brevi;
- (g) al momento dello *scheduling* calcola per ogni processo il rapporto $R = (w + s)/s$, dove w è il tempo speso in attesa del processore ed s è il tempo stimato per l'esecuzione dell'intero processo;
- (h) al momento dello *scheduling* calcola per ogni processo il rapporto $R = (w + s)/s$, dove w è il tempo speso in attesa del processore ed s è il tempo già trascorso in esecuzione;
- (i) nessuna delle affermazioni precedenti è corretta.

2.1.3 Scheduler First Come First Served (~~FCFS~~) **FIFO**

(punti: -1,4)

Lo *scheduling* del processore in ordine di arrivo (FCFS):

Stupido, non assegna priorità

- (a) non considera il processore una risorsa preilasciabile ed assegna a tutti i processi pronti uguale priorità;
- (b) considera il processore una risorsa preilasciabile ed assegna a tutti i processi pronti uguale priorità;
- (c) considera il processore una risorsa preilasciabile e non assegna a tutti i processi pronti uguale priorità;
- ☒ (d) non considera il processore una risorsa preilasciabile e non assegna a tutti i processi pronti uguale priorità;
- (e) è particolarmente adatto per i processi interattivi; No, quello è round robin
- ☒ (f) non è particolarmente adatto per i processi interattivi;
- ☒ (g) fornisce prestazioni migliori per processi lunghi che per processi brevi;
- (h) fornisce prestazioni migliori per processi brevi che per processi lunghi;
- (i) può portare a tempi di turnaround normalizzati troppo bassi per processi brevi;
- ☒ (j) può portare a tempi di *turnaround* normalizzati troppo elevati per processi brevi;
- (k) nessuna delle affermazioni precedenti è corretta.

2.1.4 Scheduling e starvation

(punti: -1,4)

In un sistema di *scheduling*, quali dei seguenti algoritmi possono causare *starvation* (attesa indefinita)?

(a) *First Come First Served* (FIFO); Fa effetto convoglio, non starvation

☒ (b) *Last Come First Served* (LIFO);

(c) *Round Robin*;

(d) *Round Robin* con quanto di tempo di durata illimitata;

☒ (e) *Shortest Process Next* (SPN); si è l'SJF

~~(f) *Shortest Remaining Time* (SRT);~~

~~(g) *Highest Response Ratio Next* (HRRN);~~

~~(h) *Lowest Response Ratio Next* (LRRN);~~

(i) l'algoritmo utilizzato dallo scheduler tradizionale di UNIX.

2.1.5 Scheduling e starvation 2

(punti: -1,4)

Il fenomeno della *starvation* (o attesa indefinita) di un processo P:

☒ (a) è tipico delle politiche di *scheduling* basate su priorità statica, infatti prima che possa essere eseguito P, potrebbero continuare a pervenire altri processi con priorità maggiore;

(b) è escluso dalle politiche di *scheduling* basate su priorità statica, infatti prima che possa essere eseguito P, il sistema non accetta altri processi con priorità maggiore;

(c) è tipico della politica di *scheduling* FIFO;

☒ (d) è escluso dalla politica di *scheduling* FIFO;

(e) può essere evitato utilizzando in P meccanismi di mutua esclusione;

~~(f) può verificarsi se P è bloccato su un semaforo di tipo forte (*strong*);~~

~~(g) può verificarsi se P è bloccato su un semaforo di tipo debole (*weak*);~~

~~(h) può essere evitato utilizzando in P meccanismi di *message passing*;~~

☒ (i) può verificarsi in caso di I/O su un disco con politica di *scheduling* LIFO;

(j) si verifica se P è coinvolto in uno stallo;

(k) si verifica se P è coinvolto in un'attesa circolare;

(l) nessuna delle affermazioni precedenti è corretta.

2.1.6 Scheduler programmabile

Un sistema operativo per macchine monoprocesso utilizza uno scheduler di processi basato su quanti di tempo e priorità. Lo scheduler opera ripetendo all'infinito un ciclo composto dai quattro passi seguenti:

- viene chiamata una subroutine definita dall'amministratore del sistema, che aggiorna le priorità di tutti i processi;
- viene determinato l'insieme dei processi che hanno il massimo valore di priorità su tutti quelli in stato di pronto;
- viene assegnato un quanto di tempo fisso di k unità temporali ad un processo scelto a caso in tale insieme che passa in esecuzione senza possibilità di prerilascio finché termina o fino allo scadere del quanto di tempo;
- al termine del quanto di tempo (o alla sua terminazione) il processo viene prerilasciato e lo scheduler ripete il ciclo.

All'atto della sottomissione di un processo, gli viene attribuita priorità 0, ed è possibile specificare la durata totale del processo.

Scrivere diverse versioni delle subroutine di aggiornamento delle priorità in modo da permettere che lo scheduler si comporti in modo da simulare le seguenti politiche:

1. *Round-Robin* (RR),
2. *First-Come-First-Served* (FCFS),
3. *Shortest Remaining Time* (SRT),

Per semplicità si assume di trascurare qualsiasi iterazione con i dispositivi di I/O e ogni altro evento che possa bloccare i processi.

Infine, si assume che nel Process Control Block (P) di ogni processo siano memorizzate le seguenti informazioni:

- $P.priority$. La priorità del processo, che sarà dinamica, modificata dalla subroutine e inizializzata a 0 al momento della creazione di P .
- $P.w$. Tempo di attesa in coda ready di P : quanto tempo P ha atteso il processore finora. Si suppone che $P.w$ viene settato a 0 quando P si aggiunge alla coda processi pronti per la prima volta e “congelato” quando P è in esecuzione. Per cui $P.w$ è il tempo netto che P passa in coda pronti, escluso il tempo d’esecuzione.
- $P.s$. Tempo d’esecuzione (o di “servizio”) totale previsto. Tale valore è inizializzato al momento della creazione di P e non più modificato.
- $P.e$. Tempo d’esecuzione di P : per quanto tempo P ha occupato il processore finora.

2.1.7 Prerilascio di un processo → DOMANDA

(punti: 8)

Illustrare in al più 70 parole cos’è il prerilascio di un processo e quali vantaggi e svantaggi presenta.

2.1.8 Esercizio scheduler Round-Robin

(punti: 6)

Considerare un insieme di cinque processi P_1, P_2, P_3, P_4, P_5 con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

Processo	Tempo di arrivo	Tempo di esecuzione
P_1	0	14
P_2	30	16
P_3	6	40
P_4	46	26
P_5	22	28

Assegnare questo insieme di processi ad un processore in base alla politica Round Robin considerando un quanto di tempo di 12 millisecondi.

Calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

2.1.9 Esercizio scheduler SRT

(punti: 6)

Considerare un insieme di sei processi P_1, P_2, P_3, P_4, P_5 e P_6 , con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

Processo	Tempo di arrivo	Tempo di esecuzione
P_1	0	10
P_2	6	6
P_3	11	15
P_4	13	6
P_5	20	2
P_6	29	9

Assegnare questo insieme di processi ad un processore in base alla politica Shortest Remaining Time.

Calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

2.1.10 Secondo esercizio di scheduler Round-Robin

(punti: 6)

Considerare un insieme di quattro processi P_1, P_2, P_3, P_4 , con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

Processo	Tempo di arrivo	Tempo di esecuzione
P_1	0	10
P_2	3	9
P_3	8	11
P_4	12	7

Assegnare questo insieme di processi ad un processore in base alla politica Round Robin considerando un quanto di tempo di 5 millisecondi.

Calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

2.1.11 Terzo esercizio di scheduler Round-Robin

(punti: 6)

Considerare un insieme di cinque processi P_1, P_2, P_3, P_4, P_5 con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

Processo	Tempo di arrivo	Tempo di esecuzione
P_1	2	20
P_2	5	7
P_3	10	13
P_4	16	7
P_5	21	7

Assegnare questo insieme di processi ad un processore in base alla politica Round Robin considerando un quanto di tempo di 6 millisecondi.

Calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

2.1.12 Esercizio di scheduler Highest Response Ratio Next

(punti: 6)

Considerare un insieme di cinque processi P_1, P_2, P_3, P_4, P_5 con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

Processo	Tempo di arrivo	Tempo di esecuzione
P_1	0	14
P_2	6	40
P_3	11	28
P_4	30	26
P_5	46	16

Assegnare questo insieme di processi ad un processore in base alla politica *Highest Response Ratio Next* e calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

2.2 Soluzioni

2.2.1 Scheduler Round-Robin

Risposte corrette: (b), (c), (f).

2.2.2 Scheduler Highest Response Ratio Next

Risposte corrette: (b), (g).

2.2.3 Scheduler First Come First Served (FCFS)

Risposte corrette: (d), (f), (g), (j).

2.2.4 Scheduling e starvation

Risposte corrette: (b), (e), (f), (h).

2.2.5 Scheduling e starvation 2

Risposte corrette: (a), (d), (g), (i).

2.2.6 Scheduler programmabile

Possibili pseudo-codice delle subroutine d'interesse sono i seguenti:

Round Robin

```
static runpriority = -1;
void Aggiorna Priorita RR ()
{
  /* forza l'ordinamento dei processi appena creati */
  1  trova insieme dei processi  $P = \{P_1, \dots, P_n\}$  tale che
      $P_i.priority == 0, 1 \leq i \leq n$ ;
  2  per ogni elemento  $P \in P$ :
  3     $P.priority = P.w$ ;

  /* mette in coda il processo che ha appena girato (se esiste) */
  4  trova il processo  $P_M$  tale che  $P_M.priority$  sia massima;
  5  se  $P_M.priority == runpriority$ :
  6     $P_M.priority = 0$ ;
     /* preserva l'ordine dei processi secondo il tempo passato in attesa
     dall'ultimo time slice ricevuto */
  7  per ogni processo  $P$ :
  8     $P.priority += k$ ;

  9  trova il processo  $P_M$  tale che  $P_M.priority$  sia massima;
  10  $runpriority = P_M.priority$ ;
}
```

Le linee 1-3 della procedura hanno la funzione di assegnare ad ogni processo ("nuovo") arrivato in coda pronti nell'ultimo time slice una priorità uguale al tempo di attesa. I processi "nuovi", infatti, sono contraddistinti dal fatto che la loro priorità è inizialmente 0. Ciò crea un ordinamento temporale tra i processi.

Tale ordinamento sarà poi conservato ad ogni time slice aumentando la priorità di tutti i processi di k unità (linee 7 e 8)

Per impedire che il processo appena eseguito, e il cui quanto di tempo è scaduto, torni immediatamente in esecuzione, le linee 4-6 azzerano la sua priorità. Infatti il processo appena eseguito può essere individuato cercando il processo P_M con priorità massima (linea 4). Tuttavia, è anche necessario assicurarsi che il processo in esecuzione nell'ultimo time slice non sia terminato, nel qual caso non starebbe più in coda pronti e il massimo trovato P_M non corrisponderebbe più al processo appena eseguito. Perciò è necessario introdurre un ulteriore controllo (linea 5) per assicurarsi che la priorità di P_M sia uguale alla priorità ($runpriority$) memorizzata nell'ultima chiamata della subroutine (linea 10).

First Come First Served

```
void Aggiorna Priorita FCFS ()
{
  Per ogni processo  $P$  in coda pronti:
     $P.priority = P.w + P.e$ ;
}
```

Shortest Remaining Time

Per quanto riguarda SRT va notato che tale politica, prevedendo un prerilascio del processore non appena dovesse eventualmente giungere un processo con priorità più alta (ovvero con un valore minore di $s - e$) di quello in esecuzione, non può essere realmente simulata da uno scheduler in cui il controllo della priorità e il relativo assegnamento del processore avviene solo allo scadere di prefissati quanti di tempo.

Un'approssimazione della politica SRT, con prerilascio possibile solo ad intervalli di tempo fissi, può essere implementata mediante la subroutine seguente:

```

void Aggiorna Priorita SRT ()
{
    Per ogni processo P in coda pronti:
         $P.priority = -(P.s - P.e)$ ;
}

```

2.2.7 Prerilascio di un processo

Il prerilascio di un processo consiste nella sottrazione temporanea dell'uso del processore ad un processo running da parte dello scheduler. Comporta operazioni aggiuntive come il salvataggio ed il ripristino dei registri di CPU, ed il rallentamento dell'esecuzione dei singoli processi, ma consente il progresso dell'esecuzione di più processi contemporaneamente e di ripartirne l'uso della CPU in base alle loro priorità.

2.2.8 Esercizio scheduler Round-Robin

Inizio	Fine	Processo
0	12	P_1
12	24	P_3
24	26	$P_1 *$
26	38	P_5
38	50	P_3
50	62	P_2
62	74	P_5
74	86	P_4
86	98	P_3
98	102	$P_2 *$
102	106	$P_5 *$
106	118	P_4
118	122	$P_3 *$
122	124	$P_4 *$

	Tin	Ts	Tout	Tta	Tatt
P1	0	14	26	26	12
P2	30	16	102	72	56
P3	6	40	122	116	76
P4	46	26	124	78	52
P5	22	28	106	84	56
		124		376	252

Tempo di turnaround medio: 75.2 ms

Tempo di attesa medio: 50.4 ms

Attenzione, non confondere il tempo di turnaround e il tempo di attesa, oppure calcolare l'attesa solo fino al primo time slice ricevuto dal processo. Il tempo di attesa è invece tutto il tempo speso dal processo nel sistema senza utilizzare il processore.

2.2.9 Esercizio scheduler Shortest Remaining Time

Inizio	Fine	Processo
0	10	$P_1 *$
10	16	$P_2 *$
16	22	$P_4 *$
22	24	$P_5 *$
24	29	P_3
29	38	$P_6 *$
38	48	$P_3 *$

	T _{in}	T _s	T _{out}	T _{ta}	T _{tatt}
P1	0	10	10	10	0
P2	6	6	16	10	4
P3	11	15	48	37	22
P4	13	6	22	9	3
P5	20	2	24	4	2
P6	29	9	38	9	0

Tempo di turnaround medio: 13.16 ms

Tempo di attesa medio: 5.16 ms

2.2.10 Secondo esercizio scheduler Round-Robin

Inizio	Fine	Processo
0	5	P_1
5	10	P_2
10	15	P_1 *
15	20	P_3
20	24	P_2 *
24	29	P_4
29	34	P_3
34	36	P_4 *
36	37	P_3 *

	T _{in}	T _s	T _{out}	T _{ta}	T _{tatt}
P1	0	10	15	15	5
P2	3	9	24	21	12
P3	8	11	37	29	18
P4	12	7	36	24	17

Tempo di turnaround medio: 22.25 ms

Tempo di attesa medio: 13 ms

2.2.11 Terzo esercizio scheduler Round-Robin

Inizio	Fine	Processo
2	8	P_1
8	14	P_2
14	20	P_1
20	26	P_3
26	27	P_2 *
27	33	P_4
33	39	P_1
39	45	P_5
45	51	P_3
51	52	P_4 *
52	54	P_1 *
54	55	P_5 *
55	56	P_3 *

	T _{in}	T _s	T _{out}
P1	2	20	54
P2	5	7	27
P3	10	13	56
P4	16	7	52
P5	21	7	55

Tempo di turnaround medio: $(244 - 54)/5 = 38ms$.

Tempo di attesa medio: $(190 - 54)/5 = 136/5 = 27.2ms$. Non essendo necessario calcolare i tempi di turnaround normalizzati, non c'è bisogno di calcolare i tempi di turnaround dei singoli processi.

2.2.12 Esercizio scheduler Highest Response Ratio Next

Inizio	Fine	Processo
0	14	P_1 *
14	54	P_2 *
54	82	P_3 *
82	98	P_5 *
98	124	P_4 *

	Tin	Tout	Tta	Tatt
P1	0	14	14	0
P2	6	54	48	8
P3	11	82	71	43
P4	30	124	94	68
P5	46	98	52	36

Tempo di turnaround medio: $279/5 = 55.8ms$.

Tempo di attesa medio: $155/5 = 31ms$.

Chapter 3

Gestione della memoria

3.1 Testi

3.1.1 Translation Lookaside Buffer

(punti: -1,4)

In un sistema di memoria a paginazione, il *Translation Lookaside Buffer* (TLB):

- ☒ (a) è una cache che si trova all'interno della CPU; Non è giusto quel che è giusto ma è giusto quel che piace
- ☐ (b) è un buffer allocato nella memoria RAM del calcolatore;
- ☐ (c) velocizza la traduzione di indirizzi fisici in indirizzi virtuali;
- ☒ (d) velocizza la traduzione di indirizzi virtuali in indirizzi fisici;
- ☐ (e) velocizza l'identificazione delle pagine il cui contenuto è stato spostato su disco;
- ☐ (f) velocizza l'identificazione dei frame il cui contenuto è stato spostato su disco;
- ☐ (g) contiene la tabella delle pagine del processo in esecuzione;
- ☐ (h) contiene la tabella delle pagine di tutti i processi in esecuzione;
- ☐ (i) contiene le informazioni relative alle pagine usate più di recente dal processo in esecuzione;
- ☒ (j) contiene le informazioni relative alle pagine usate più di recente dal processore, indipendentemente dal processo a cui appartengono;
- ☐ (k) viene aggiornata automaticamente dall'*hardware* ogni volta che si accede ad una pagina non descritta in TLB; vecchio
- ☒ (l) viene aggiornata dal sistema operativo, invocato ogni volta che si accede ad una pagina non descritta in TLB.

3.1.2 Gestione della memoria con segmentazione

(punti: -1,4)

La gestione della memoria con segmentazione:

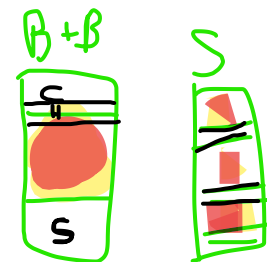
- ☐ (a) è una generalizzazione della paginazione;
- ☐ (b) è una generalizzazione del partizionamento statico;
- ☒ (c) è una generalizzazione del partizionamento dinamico; base e bound
- ☒ (d) è soggetta a frammentazione esterna;
- ☐ (e) è soggetta a frammentazione interna; no paginazione quella
- ☐ (f) non è soggetta ad alcuna frammentazione; si, frammentazione esterna
- ☐ (g) peggiora l'organizzazione interna dei processi, forzando la separazione di blocchi di codice e di dati;
- ☒ (h) razionalizza l'organizzazione interna dei processi, consentendo la separazione di blocchi di codice e di dati;
- ☐ (i) aumenta il consumo di memoria, forzando in ogni processo la suddivisione di blocchi di codice e di dati in segmenti differenti;
- ☐ (j) aumenta il consumo di memoria, forzando più processi a condividere blocchi di codice e di dati;
- ☒ (k) riduce il consumo di memoria, consentendo a più processi di condividere blocchi di codice e di dati; bit di protezione
- ☐ (l) non è descritta correttamente da nessuna delle affermazioni precedenti.

3.1.3 Peculiarità dei sistemi di gestione della memoria

(punti: -1,4)

Nella gestione della memoria:

- ☒ (a) il sistema a partizionamento fisso soffre di frammentazione interna;
- ☐ (b) il sistema a partizionamento fisso soffre di frammentazione esterna;
- ☐ (c) il sistema a partizionamento dinamico soffre di frammentazione interna; perchè no?
- ☒ (d) il sistema a partizionamento dinamico soffre di frammentazione esterna; ?
- ☐ (e) il sistema a segmentazione soffre di frammentazione interna;
- ☒ (f) il sistema a segmentazione soffre di frammentazione esterna;
- ☒ (g) il sistema che combina segmentazione e paginazione soffre di frammentazione interna;
- ☐ (h) il sistema che combina segmentazione e paginazione soffre di frammentazione esterna; perchè no?
- ☐ (i) tutte le affermazioni precedenti sono corrette.



3.1.4 Gestione della memoria, il fenomeno della frammentazione interna

(punti: -1,4)

Nella gestione della memoria, il fenomeno della frammentazione interna:

- ☐ (a) in caso di partizionamento statico, è tanto più rilevante quanto più la lunghezza media dei programmi è grande rispetto alla dimensione delle partizioni;
- ☒ (b) in caso di paginazione, è tanto meno rilevante quanto più la lunghezza media dei programmi è grande rispetto alla dimensione della pagina;

- ☒ (a) riduce la quantità di memoria utilizzabile;
- (d) non riduce la quantità di memoria utilizzabile;
- (e) rimuove il vincolo della contiguità dello spazio fisico in memoria centrale;
- (f) mantiene il vincolo della contiguità dello spazio fisico in memoria centrale;
- (g) nessuna delle affermazioni precedenti è corretta.

3.1.5 Politiche di sostituzione delle pagine

(punti: -1,4)

Quali delle seguenti affermazioni sulle politiche di sostituzione delle pagine sono corrette?

- (a) la politica LRU causa in ogni caso meno *page fault* della politica del *Clock*;
- (b) la politica LRU causa in ogni caso più *page fault* della politica del *Clock*;
- (c) la politica del *Clock* non causa mai meno *page fault* della politica LRU;
- (d) la politica del *Clock* non causa mai più *page fault* della politica LRU;
- (e) la politica FIFO non causa mai meno *page fault* della politica LRU;
- (f) la politica FIFO non causa mai più *page fault* della politica LRU;
- ☒ (g) le strutture dati necessarie per implementare la politica LRU richiedono più memoria per pagina gestita rispetto alla politica del *Clock*;
- (h) le strutture dati necessarie per implementare la politica LRU richiedono meno memoria per pagina gestita rispetto alla politica del *Clock*;
- (i) la politica FIFO equivale ad una politica LRU con zero bit di informazione sull'uso di ogni pagina; ?
- (j) la politica FIFO equivale ad una politica LRU con trentadue bit di informazione sull'uso di ogni pagina;
- ☒ (k) la politica FIFO equivale ad una politica del *Clock* con zero bit di informazione sull'uso di ogni pagina;
- (l) la politica FIFO equivale ad una politica del *Clock* con trentadue bit di informazione sull'uso di ogni pagina;
- (m) nessuna delle affermazioni precedenti è corretta.

3.1.6 Obiettivi nella gestione della memoria

(punti: 7)

Illustrare in al più 70 parole gli obiettivi (o requisiti da soddisfare) della gestione della memoria.

3.1.7 Algoritmo del Clock

(punti: 9)

Illustrare in al più 100 parole l'algoritmo di sostituzione del *clock* (o *second chance*), ed i suoi vantaggi e svantaggi rispetto all'algoritmo *Least Recently Used* (LRU).

3.1.8 Sequenza di accessi in memoria con algoritmi LRU e Clock

Considerare la seguente stringa di riferimenti alla memoria di un processo in un sistema con memoria virtuale:

S= 5 7 10 15 19 23 10 7 5 19 15 14 17 19 15 16 18 17 14 23

(a) (punti: 3)

Illustrare il comportamento dell'algoritmo LRU di sostituzione delle pagine per una memoria fisica di 5 blocchi. Calcolare il numero di *page fault* che si verificano.

(b) (punti: 3)

Illustrare il comportamento dell'algoritmo del *Clock* di sostituzione delle pagine per una memoria fisica di 5 blocchi. Calcolare il numero di *page fault* che si verificano.

3.1.9 Traduzione indirizzi con paginazione

Considerare un sistema di traduzione da indirizzamento logico a indirizzamento fisico realizzato mediante paginazione. Lo spazio logico di un programma è costituito da un massimo di 64 byte, suddivise in pagine da 4 byte. La memoria fisica è costituita da 256 byte.

(a) Da quanti bit sono costituiti gli indirizzi logici e gli indirizzi fisici?

- (b) Da quanti bit sono costituiti i numeri di pagina?
 (c) Da quanti bit sono costituiti i numeri di *frame*?
 (d) Ad un dato istante, la tabella delle pagine (*page table*) di un processo è la seguente:

Numero pagina logica	Numero pagina fisica
0	12
1	1
2	17
3	62
4	11
5	16
6	61
7	12

Tradurre in indirizzi fisici i seguenti indirizzi logici: 0, 2, 4, 9, 19, 11, 22, 32, 30, 26, 23, 36.

3.1.10 Secondo esercizio su Traduzione indirizzi con paginazione

Considerare un sistema di traduzione da indirizzamento logico a indirizzamento fisico realizzato mediante paginazione. Lo spazio logico di un programma è costituito da un massimo di 128 byte, suddivise in pagine da 8 byte. La memoria fisica è costituita da 256 byte.


- (a) Da quanti bit sono costituiti gli indirizzi logici e gli indirizzi fisici?
 (b) Da quanti bit sono costituiti i numeri di pagina?
 (c) Da quanti bit sono costituiti i numeri di *frame*?
 (d) Ad un dato istante, la tabella delle pagine (*page table*) di un processo è la seguente:

Numero pagina logica	Numero pagina fisica
0	7
1	30
2	25
3	15
4	12
5	11
6	27
7	20
8	10
9	31
10	3
11	6
12	0
13	29
14	17
15	13

Tradurre in indirizzi fisici i seguenti indirizzi logici: 0, 3, 7, 9, 19, 11, 21, 30, 120, 64.

3.1.11 Traduzione degli indirizzi con segmentazione

Considerare un sistema con memoria virtuale e segmentazione con parola da un byte. Supponendo di avere indirizzi logici di 10 bit, i cui primi 4 più significativi indicano il numero di segmento, dire:

- (a) qual è la massima grandezza possibile per un segmento; 64 byte $1024/16$ 
- (b) di quanti segmenti al più può essere composto un processo. $2^4 = 16$

Supponendo che il processo attualmente in esecuzione sia composto da 5 segmenti S_0, S_1, S_2, S_3 ed S_4 e che, a un dato istante, la sua tabella dei segmenti sia la seguente:

Numero di segmento	Lunghezza	Base
0	10	200
1	20	100
2	6	252
3	32	720
4	32	683

tradurre in indirizzi fisici i seguenti indirizzi logici: 9, 132, 79, 64, 259, 135, 320.

3.2 Soluzioni

3.2.1 Translation Lookaside Buffer

Risposte corrette: (a), (d), (j), (l).

3.2.2 Gestione della memoria con segmentazione

Risposte corrette: (c), (d), (h), (k).

3.2.3 Peculiarità dei sistemi di gestione della memoria

Risposte corrette: (a), (d), (f), (g).

Il sistema di gestione della memoria a partizionamento fisso consiste nel dividere la memoria in partizioni statiche ed assegnare a ciascun processo una partizione di dimensione uguale o superiore. Se ad un processo viene assegnato più spazio di quello necessario si ha spreco di spazio all'interno della partizione (frammentazione interna), mentre non si ha mai spazio "spreco" fuori dalle partizioni (tutta la memoria viene divisa in partizioni). Quindi l'affermazione (a) è esatta mentre è errata l'affermazione (b). Nel **partizionamento dinamico le partizioni vengono invece create dinamicamente** (nella fase di caricamento del processo) una partizione della dimensione corrispondente a quella necessaria al processo. In questa maniera non si ha frammentazione interna. Tuttavia, si ha **frammentazione esterna** perchè se viene "liberata" una **partizione di dimensione minore a quella richiesta da un nuovo processo, questo spazio non può essere direttamente utilizzato**. Quindi, l'affermazione (c) è falsa, mentre è corretta l'affermazione (d). Nella gestione a segmentazione è il processo ad essere diviso in segmenti, che vengono poi caricati in memoria non necessariamente in spazi contigui. Attenzione: non è richiesto che tutti i segmenti di tutti i programmi siano della stessa lunghezza. Comunque, nessuno spazio all'interno del segmento è sprecato: la risposta (e) è falsa. Tuttavia, lo spazio di memoria liberato corrispondente ad un segmento può essere direttamente riassegnato ad un segmento di un nuovo processo solo se è di dimensione sufficiente: **la risposta (f) è quindi vera**. Combinare la segmentazione con la paginazione implica frammentazione interna: spazio delle pagine potrebbe essere sprecato se non assegnato per intero ad un segmento. **La risposta (g) è quindi corretta**. Invece, l'allocazione dei segmenti, il loro numero, etc... non hanno nulla a che vedere con la frammentazione esterna. Infatti, se si assembla ogni segmento da pagine di memoria, non c'è più necessità che il segmento sia un insieme contiguo di indirizzi in memoria fisica (come invece richiesto ad esempio nel partizionamento), quindi non ci sarà più frammentazione esterna. La stessa cosa avviene in generale nel sistema a paginazione. Ne segue che l'affermazione (h) è falsa.

3.2.4 Gestione della memoria, il fenomeno della frammentazione interna

Risposte corrette: (b), (c).

3.2.5 Politiche di sostituzione delle pagine

Risposte corrette: (g), (k).

3.2.6 Obiettivi nella gestione della memoria

La gestione della memoria da parte del sistema operativo intende soddisfare in generale i seguenti requisiti:

- possibilità di rilocalizzazione (non è possibile conoscere in anticipo gli altri programmi che saranno presenti in memoria durante l'esecuzione di un processo, inoltre, quando un processo viene spostato - *swap* - su disco è difficile garantire che sarà ricaricato in memoria nella stessa posizione);
- protezione delle aree di memoria assegnate ai processi;
- possibilità di condividere le aree di memoria assegnate,

- organizzazione logica (modularizzazione del codice e dell'eseguibile)
- mascheramento della particolare organizzazione fisica della memoria

Note:

Attenzione a non trascurare gli aspetti dell'organizzazione logica e dell'astrazione dalla struttura fisica. Legare il problema della rilocalizzazione solo allo swap di processi sarebbe limitativo: il problema si verificherebbe anche senza swapping, per il fatto di richiamare in occasioni diverse lo stesso eseguibile da disco.

3.2.7 Algoritmo del Clock

Note: chiarire che il bit di uso viene posto ad 1 ad **ogni accesso** a dati contenuti in una pagina. Non confondere un accesso alla memoria di una pagina da parte del processore con il caricamento da disco di una pagina non in memoria.

Attenzione, l'algoritmo del clock non viene invocato ad ogni accesso in memoria (sarebbe disastroso), ma solo quando e' necessario caricare da disco una pagina al posto di una di quelle gia' caricate. Non e' necessario spiegare cosa avviene quando l'algoritmo e' invocato e tutte le pagine hanno bit di uso ad 1: e' inutile, e' un'ovvia conseguenza dell'algoritmo.

E' errato affermare che l'algoritmo del clock fa miglior uso del principio di localita' rispetto ad LRU: l'algoritmo LRU fa il miglior uso possibile del principio di localita' temporale, ed infatti l'algoritmo del clock e' in genere inferiore come comportamento, anche se, essendo tutti e due sub-ottimi, puo' occasionalmente fornire risultati migliori. Il vero vantaggio dell'algoritmo del clock e' il suo ridottissimo onere computazionale, che lo fa preferire in quasi tutti i sistemi operativi.

3.2.8 Sequenza di accessi in memoria con algoritmi LRU e Clock

(a) LRU

	5	7	10	15	19	23	10	7	5	19	15	14	17	19	15	16	18	17	14	23
F_0	5	5	5	5	5	23	23	23	23	23	15	15	15	15	15	15	15	15	15	23
F_1		7	7	7	7	7	7	7	7	7	7	7	17	17	17	17	17	17	17	17
F_2			10	10	10	10	10	10	10	10	10	14	14	14	14	14	18	18	18	18
F_3				15	15	15	15	15	5	5	5	5	5	5	5	16	16	16	16	16
F_4					19	19	19	19	19	19	19	19	19	19	19	19	19	19	14	14
page fault	1	2	3	4	5	6			7		8	9	10			11	12		13	14

(a) Clock

	5	7	10	15	19	23	10	7	5	19	15	14	17	19
F_0	5*	5*	5*	5*	>5*	23*	23*	23*	23*	23*	23	23	>23	19*
F_1	>	7*	7*	7*	7*	>7	>7	>7*	7	7	15*	15*	15*	>15*
F_2		>	10*	10*	10*	10	10*	10*	10	10	>10	14*	14*	14*
F_3			>	15*	15*	15	15	15	5*	5*	5*	>5*	5	5
F_4				>	19*	19	19	19	>19	>19*	19	19	>17*	17*
page fault	1	2	3	4	5	6			7		8	9	10	11

	15	16	18	17	14	23
F_0	19*	19*	19	19	19	23*
F_1	>15*	15	18*	18*	18*	>18*
F_2	14*	14	>14	>14	>14*	14
F_3	5	16*	16*	16*	16*	16
F_4	17*	>17*	17	17*	17*	17
page fault		12	13			14

Attenzione, i page fault iniziali (5 in questo caso) devono essere considerati nel conteggio totale ed indicati esplicitamente nella tabella. Nota: non bisogna fare confusione al momento in cui avviene un page fault nell'algoritmo del *Clock*. Il frame da cui riprende la ricerca e' il successivo a quello nel quale e' avvenuto l'ultimo caricamento, e **non** il successivo a quello acceduto per ultimo!

3.2.9 Traduzione indirizzi con paginazione

(a) La traccia specifica che lo spazio logico di un programma e' al massimo composto da 64 byte. Ciò significa che l'indirizzo logico deve poter codificare 64 diverse posizioni (da 0 a 63). Abbiamo bisogno quindi di una stringa binaria che possa rappresentare i numeri da 0 a 63. La lunghezza di questa stringa ci dà il numero di bit dell'indirizzo logico (e un ragionamento analogo vale per l'indirizzo fisico).

Esempio. Se il massimo spazio logico indirizzabile fosse composto da due soli byte (rispettivamente, indirizzo 0 e indirizzo 1), allora sarebbe sufficiente un numero binario di una sola cifra (bit) per indirizzare entrambe le posizioni: con un bit, infatti, rappresento sia il numero 0 che il numero 1. Se lo spazio logico fosse di 4 byte (indirizzi: 0, 1, 2 e 3), avremmo bisogno di un indirizzo a due cifre. Se lo spazio indirizzabile e' di N byte, c'è bisogno di $\lceil \log_2(N) \rceil$ bit, dove $\lceil X \rceil$ e' la funzione che approssima X per eccesso all'intero maggiore più vicino (e.g., $\lceil 0.2 \rceil = 1$).

Quindi, nel caso in esame, l'indirizzo logico e' composto da $\log_2(64) = 6$ bit e l'indirizzo fisico da $\log_2(256) = 8$ bit.

(b) Quante pagine da 4 byte ci sono in uno spazio da 64 byte? $64/4 = 16$. Per un ragionamento analogo al punto (a), devo poter rappresentare 16 posizioni, per cui ho: $\log_2(16) = 4$.

(c) Idem: $256/4 = 64$ (ricordiamoci che la grandezza di una pagina è la stessa della grandezza di un frame); $\log_2(64) = 6$.

(d)

```
0 = 0*4 + 0 -> 12*4 + 0 = 48
2 = 0*4 + 2 -> 12*4 + 2 = 50
4 = 1*4 + 0 -> 1*4 + 0 = 4
9 = 2*4 + 1 -> 17*4 + 1 = 69
19 = 4*4 + 3 -> 11*4 + 3 = 47
11 = 2*4 + 3 -> 17*4 + 3 = 71
22 = 5*4 + 2 -> 16*4 + 2 = 66
32 = 8*4 + 0 -> ERRORE/SEGMENTATION FAULT
30 = 7*4 + 2 -> 12*4 + 2 = 50
26 = 6*4 + 2 -> 61*4 + 2 = 246
23 = 5*4 + 3 -> 16*4 + 3 = 67
36 = 9*4 + 0 -> ERRORE/SEGMENTATION FAULT
```

3.2.10 Secondo esercizio su Traduzione indirizzi con paginazione

a) indirizzi logici 7 bit, indirizzi fisici 8 bit

b) 4

c) 5

d)

```
0 = 0*8 + 0 -> 7*8 + 0 = 56
3 = 0*8 + 3 -> 7*8 + 3 = 59
7 = 0*8 + 7 -> 7*8 + 7 = 63
9 = 1*8 + 1 -> 30*8 + 1 = 241
19 = 2*8 + 3 -> 25*8 + 3 = 203
11 = 1*8 + 3 -> 30*8 + 3 = 243
21 = 2*8 + 5 -> 25*8 + 5 = 205
30 = 3*8 + 6 -> 15*8 + 6 = 126
120 = 15*8 + 0 -> 13*8 + 0 = 104
64 = 8*8 + 0 -> 10*8 + 0 = 80
```

3.2.11 Traduzione degli indirizzi con segmentazione

- (a) $2^6 = 64$.

- (b) $2^4 = 16$.

- (c)

- $9 \rightarrow (0, 9) \rightarrow 200 + 9 = 209$;
- $132 \rightarrow (2, 4) \rightarrow 252 + 4 = 256$;
- $79 \rightarrow (1, 15) \rightarrow 100 + 15 = 115$;
- $64 \rightarrow (1, 0) \rightarrow 100 + 0 = 100$;
- $259 \rightarrow (4, 3) \rightarrow 683 + 3 = 686$;
- $135 \rightarrow (2, 7) \rightarrow \text{segmentation fault}$;
- $320 \rightarrow (5, 0) \rightarrow \text{invalid segment}$;

Chapter 4

Gestione I/O

4.1 Testi

4.1.1 Proprietà della tecnica del Buffering

(punti: -1,4)

La tecnica del *buffering*:

- (a) è applicabile esclusivamente ai dispositivi di I/O a blocchi (*block – oriented*);
- (b) è applicabile esclusivamente ai dispositivi di I/O a carattere (*stream – oriented*);
- (c) è applicabile a dispositivi di I/O sia a blocchi (*block – oriented*) che a carattere (*stream – oriented*);
- (d) è applicabile solo all'*input*;
- (e) è applicabile solo all'*output*;
- (f) può velocizzare l'esecuzione dei programmi, consentendo la sovrapposizione dell'elaborazione dei dati già ricevuti con l'*input* dei dati successivi;
- (g) può rallentare l'esecuzione dei programmi, non essendo realmente possibile la sovrapposizione dell'elaborazione dei dati già ricevuti con l'*input* dei dati successivi;
- (h) può velocizzare l'esecuzione dei programmi, consentendo la sovrapposizione dell'*output* dei dati già prodotti con l'elaborazione dei risultati successivi;
- (i) può rallentare l'esecuzione dei programmi, non essendo realmente possibile la sovrapposizione dell'*output* dei dati già prodotti con l'elaborazione dei risultati successivi;
- (j) consente l'esecuzione senza interruzioni di processi che richiedono o producono dati a velocità medie molto superiori a quelle sostenibili dai dispositivi di I/O;
- (k) richiede l'utilizzo di *buffer* circolari;
- (l) consiste nell'anticipare l'*input* (ritardare l'*output*) dei dati rispetto alla richiesta da parte di un processo, utilizzando aree di memoria esterne a quest'ultimo come appoggio temporaneo per i dati stessi.

4.1.2 DMA

(punti: -1,4)

In un elaboratore, il DMA (*Direct Memory Access*):

- (a) è una tecnica che demanda l'I/O ad un apposito modulo ~~software~~; è un chip cazo
- ~~(b)~~ è una tecnica che demanda l'I/O ad un apposito modulo *hardware*; è un chip, hardware
- (c) è una tecnica che evita il ricorso a qualsiasi *interrupt*; pochi deve comunque farli
- ~~(d)~~ è una tecnica che utilizza un numero minimo di *interrupt*;
- (e) è l'attributo dei segmenti di memoria che possono essere acceduti da ogni processo;
- (f) è lo stato a cui passa un processo uscendo dallo stato *Suspend*.

4.1.3 Obiettivi del disk scheduling

(punti: 9)

Illustrare in al più 100 parole gli obiettivi del *disk scheduling*, le modalità generali con cui li persegue ed uno tra i possibili algoritmi utilizzati.

4.1.4 Buffering

Considerare un programma che accede un singolo dispositivo di I/O e comparare la modalità di accesso con buffer alla modalità senza buffer. Qual è il massimo guadagno possibile nel tempo d'esecuzione? Dimostrare le affermazioni fatte.

4.1.5 SSTF

Considerare una politica di Disk Scheduling gestita tramite l'algoritmo Shortest Service Time First. Si supponga, per semplicità, un tempo di seek lineare col numero di tracce attraversate dalla testina e un ritardo rotazionale e un tempo di trasferimento dati approssimabili con 0. Il tempo di seek è di una traccia al millisecondo, la testina al tempo 0 si trova sulla traccia numero 50 e il disco è composto da 200 tracce (da 0 a 199). Una volta che viene avviata una richiesta di I/O al disco non è possibile interromperla, per cui le scelte operate dallo scheduler avvengono soltanto dopo il completamento di una richiesta di I/O. Si supponga, infine, che, se non ci sono richieste pendenti, la testina rimanga ferma sull'ultima traccia visitata.

Data la seguente tabella, che indica i tempi di arrivo in millisecondi di varie richieste di I/O su disco e le relative tracce, fornire:

- (a) la risultante sequenza con cui esse vengono soddisfatte dallo scheduler e i relativi tempi;
- (b) il tempo d'attesa medio di tutte le richieste.

Numero richiesta	Traccia	Tempo d'arrivo
1	25	1
2	60	5
3	180	15
4	100	30
5	80	40
6	70	70
7	60	90
8	30	140

4.1.6 SCAN

Considerare una politica di Disk Scheduling gestita tramite l'algoritmo SCAN. Si supponga, per semplicità, un tempo di seek lineare col numero di tracce attraversate dalla testina e un ritardo rotazionale e un tempo di trasferimento dati approssimabili con 0. Il tempo di seek è di una traccia al millisecondo, la testina al tempo 0 si trova sulla traccia numero 50 e il disco è composto da 200 tracce (da 0 a 199). Una volta che viene avviata una richiesta di I/O al disco non è possibile interromperla, per cui le scelte operate dallo scheduler avvengono soltanto dopo il completamento di una richiesta di I/O. Si supponga, infine, che, se non ci sono richieste pendenti, la testina rimanga ferma sull'ultima traccia visitata e che la direzione di scansione venga invertita quando non ci sono ulteriori richieste pendenti nella direzione attuale.

Con riferimento alla tabella 1, che indica i tempi di arrivo in millisecondi di varie richieste di I/O su disco e le relative tracce, fornire:

- (a) la risultante sequenza con cui esse vengono soddisfatte dallo scheduler e i relativi tempi;
- (b) il tempo di attesa medio di tutte le richieste.

Tabella 1

Numero richiesta	Traccia	Tempo d'arrivo
1	70	2
2	65	8
3	80	20
4	75	21
5	120	29
6	60	31
7	90	38
8	80	100

4.2 Soluzioni

4.2.1 Proprietà della tecnica del Buffering

Risposte corrette: (c), (f), (h), (l).

La tecnica del *buffering* è esattamente definita dall'affermazione (l), che è quindi vera insieme alle affermazioni (f) ed (h). Segue che le affermazioni (d), (e), (g) ed (i) sono false. Anche se con accorgimenti differenti, il *buffering* può essere applicato sia a dispositivi *block – oriented* che *stream – oriented*: è vera l'affermazione (c) mentre sono false (a) e (b).

Lo scopo del *buffering* è proprio quello di "smussare" i picchi di richieste I/O riducendo quindi il numero di volte in cui il processo rimane in attesa dell'I/O. Tuttavia, nessun buffer può impedire (all'infinito) che il processo si blocchi per I/O se la velocità di richiesta del processo è mediamente superiore rispetto alla velocità con cui l'I/O riesce invece a servirlo. Per questo motivo l'affermazione (j) non è corretta.

Infine, l'affermazione (k) è falsa perchè i *buffer* circolari sono solo una delle possibili schemi di implementazione del *buffering*.

4.2.2 DMA

Risposte corrette: (b), (d).

4.2.3 Obiettivi del disk scheduling

Nella risposta si deve dire che l'obiettivo del disk scheduling è quello di ridurre il tempo di accesso medio a disco, ed è perseguito tramite la riduzione della componente dominante del tempo di accesso: il tempo di seek del braccio. Nel caso si illustri la politica C-SCAN sia chiaro che: assegnato un movimento del braccio, qualunque esso sia, le richieste vengono servite compatibilmente con la posizione raggiunta. Una richiesta per una traccia già superata dovrà attendere finché il braccio non ci ritorna sopra.

4.2.4 Buffering

Il guadagno massimo ottenibile è un dimezzamento dei tempi d'esecuzione, che si raggiunge quando il tempo di computazione (C) del programma è pari al tempo di lettura dei dati (T).

Dim.

Il tempo d'esecuzione senza buffer (SB) è dato dalla somma del tempo di lettura e del tempo di computazione: $SB = T + C$. Il tempo d'esecuzione con buffer (B) è invece dato da: $B = \max\{T, C\} + M$, dove M è il tempo di trasferimento dei dati dal buffer allo spazio del processo. Essendo $M \ll T$ possiamo approssimare $M = 0$. Da ciò si deduce che:

- Caso $C = T$. $SB/B = \frac{C+T}{\max\{T,C\}} = \frac{T+T}{T} = 2$.
- Caso $C > T$. $SB/B = \frac{C+T}{\max\{T,C\}} = \frac{C+T}{C} < \frac{C+C}{C} = 2$.
- Caso $C < T$. $SB/B = \frac{C+T}{\max\{T,C\}} = \frac{C+T}{T} < \frac{T+T}{T} = 2$.

4.2.5 SSTF

(a) Le coppie di valori (i, t_i) in ogni riga della tabella seguente indicano l'istante di tempo (t_i) con cui la i -esima richiesta viene soddisfatta:

Numero richiesta	Tempo di completamento
1	26
2	61
5	81
6	91
7	101
4	141
8	211
3	361

(b) Esaminando la tabella seguente, che riporta i tempi di attesa di ogni richiesta (= tempo di completamento - tempo di arrivo), il tempo d'accesso medio è: 85.25.

Numero richiesta	Tempo d'attesa
1	25
2	56
3	346
4	111
5	41
6	21
7	11
8	71

4.2.6 SCAN

(a) Le coppie di valori (i, t_i) in ogni riga della tabella seguente indicano l'istante di tempo (t_i) con cui la i -esima richiesta viene soddisfatta:

Numero richiesta	Tempo di completamento
1	22
4	27
3	32
5	72
7	102
8	112
2	127
6	132

(b) Esaminando la tabella seguente, che riporta i tempi di attesa di ogni richiesta (= tempo di completamento - tempo di arrivo), il tempo d'accesso medio è: 47.125.

Numero richiesta	Tempo d'attesa
1	20
2	119
3	12
4	6
5	43
6	101
7	64
8	12

Chapter 5

File System

5.1 Testi

5.1.1 Proprietà degli i-node

(punti: -1,4)

La struttura *i-node* è utilizzata nel *file system* di Unix e dei sistemi operativi derivati per descrivere i *file*. Quali delle seguenti affermazioni sono corrette?

- (a) l'*i-node* di un *file* contiene, tra le informazioni che lo descrivono, il nome;
- ☒ (b) l'*i-node* di un *file* non contiene, tra le informazioni che lo descrivono, il nome; non contiene il nome HR
- (c) l'*i-node* contiene la lista degli indici di tutti i blocchi che fanno parte del *file*;
- (d) l'*i-node* non contiene la lista degli indici di tutti i blocchi che fanno parte del *file*, elencati invece nella FAT;
- (e) l'*i-node* è codificato in forma di *bitmap*;
- (f) l'*i-node* è codificato in forma di ~~*byte-map*~~;
- ☒ (g) l'*i-node* facilita, per la sua compattezza, la conservazione in RAM delle informazioni relative ai *file* aperti;
- (h) l'*i-node* complica, per la sua struttura ad albero, la conservazione in RAM delle informazioni relative ai *file* aperti;
- (i) nessuna.

5.1.2 Struttura degli i-node

Illustrare in al più 70 parole la struttura dell'*i-node* ed i vantaggi di questo sistema di descrizione di un *file*.

5.2 Soluzioni

5.2.1 Proprietà degli i-node

Risposte corrette: (b), (g).

5.2.2 Struttura degli i-node

L'*i-node* contiene tutte le informazioni relative ad un file (proprietario, permessi di accesso, dimensione, tempi degli ultimi accessi e modifiche, ...) con l'eccezione del nome del file stesso. Inoltre contiene l'elenco dei blocchi del disco che contengono i dati del file. I primi dieci blocchi sono elencati direttamente, quelli successivi tramite tre riferimenti. I successivi sono indicati progressivamente tramite un sistema ad indicizzazione indiretta semplice, doppia e tripla: per ognuno di questi l'*i-node* contiene l'indice del blocco radice dell'indicizzazione. Il vantaggio principale dell'*i-node* è la sua compattezza, che consente facilmente il caching di tutte le informazioni pertinenti ad un file in corso di lettura o modifica.

Note: non serve esordire con la definizione di *i-node* (inutile, dato che è specificato nella domanda). I vantaggi vanno specificati con chiarezza.

Attenzione, fare riferimento alla massima dimensione di un file è un dato poco significativo perché dipende parametricamente dalla dimensione dei blocchi del file system.

Attenzione a non fare confusione nell'uso della parola "file": essa indica il contenuto di un file e non il nome (etichetta arbitraria aggiunta per nostra comodità').

Infine, una quantità sorprendente di studenti, nel rispondere a tale domanda in un compito d'esame, ha sostenuto un concetto espresso più o meno così: "Molti file possono essere associati allo stesso *i-node*, ma un file può essere controllato da un solo *i-node*". Mi pare che l'evidente incongruenza linguistica della affermazione nasca da una grave confusione nell'uso della parola "file", usata sia per riferirsi al nome dello stesso che al suo contenuto. È ovvio che è valida solo la seconda corrispondenza essendo il nome un'etichetta arbitraria aggiunta per nostra comodità'.

Chapter 6

Concorrenza

6.1 Testi

6.1.1 Semafori

(punti: -1,4)

Quali delle seguenti affermazioni sono vere?

- (a) in un semaforo *weak* i processi in coda vengono sbloccati nell'ordine di arrivo;
- ☒ (b) in un semaforo *weak* i processi in coda vengono sbloccati in un ordine non specificato;
- ☒ (c) in un semaforo *strong* i processi in coda vengono sbloccati nell'ordine di arrivo;
- (d) in un semaforo *strong* i processi in coda vengono sbloccati in un ordine non specificato;
- (e) il meccanismo dei semafori non è che un modo alternativo per forzare la mutua esclusione tra processi;
- ☒ (f) il meccanismo dei semafori oltre a consentire la mutua esclusione tra processi, supporta forme più generali di sincronizzazione tra processi;
- ☒ (g) il meccanismo della comunicazione consente anche la sincronizzazione tra processi;
- (h) il meccanismo della comunicazione non consente la sincronizzazione tra processi;
- (i) nessuna delle affermazioni precedenti è corretta.

6.1.2 Mutua esclusione

(punti: -1,4)

La mutua esclusione:

- ☒ (a) garantisce che determinate risorse siano accedute da un solo processo alla volta;
- (b) è causa di *race condition* su variabili condivise;
- (c) è causa di attesa indefinita (*starvation*);
- ☒ (d) è utilizzabile per evitare *race condition* su variabili condivise;
- (e) richiede necessariamente l'utilizzo dei semafori;
- (f) richiede necessariamente l'utilizzo di comunicazioni sincrone;
- (g) richiede necessariamente l'utilizzo di istruzioni hardware specifiche, senza le quali non può essere implementata.

6.1.3 Sezione critica

(punti: -1,4)



Una *sezione critica*:

- (a) è una risorsa non condivisibile contesa tra due o più processi; è un pezzo di codice gesu cammello
- (b) è una risorsa condivisibile contesa tra due o più processi;
- ☒ (c) è una porzione di programma nella quale si accede ad una risorsa non condivisibile contemporaneamente da più processi;
- ☒ (d) è una porzione di programma nella quale può verificarsi una *race condition* tra processi;
- (e) è la porzione di programma che determina la velocità di avanzamento complessiva di tutto il processo;
- (f) è la porzione di programma che il processo deve ad ogni costo eseguire, indipendentemente da quanto avviene nell'esecuzione del resto del codice;
- (g) può contenere non più di uno ~~statement~~;
- (h) può contenere non più di due ~~statement~~;
- (i) può contenere non più di tre ~~statement~~;
- (j) può contenere più di uno ~~statement~~ purché non si tratti di chiamate di sistema;
- (k) può contenere più di uno *statement* a condizione che ognuno sia atomico.

6.1.4 Problema lettori-scrittori

(punti: -1,4)

Il problema lettori-scrittori:

- ☒ (a) può essere risolto utilizzando la mutua esclusione;
- ☒ (b) può essere risolto utilizzando i semafori;
- ☒ (c) può essere risolto utilizzando le comunicazioni (scambio di messaggi); Non l'abbiamo FAT hahahah
- (d) può essere risolto utilizzando gli *i-node*; 
- (e) può essere risolto utilizzando la FAT; 
- (f) è un caso particolare del problema dei filosofi a cena;
- (g) è un caso particolare del problema del negozio del barbiere;
- (h) è un caso particolare del problema produttore-consumatore.

6.1.5 ~~Comunicazione tra processi tramite scambio di messaggi~~

(punti: 10)

Discutere in al più 100 parole il meccanismo della comunicazione tra processi basato su scambio di messaggi.

6.1.6 Race condition

Considerare il seguente pseudo-codice:

```
const int n = 20;
int conta;

void incrementa() {
    int i;

    for(i = 0; i < n; ++i) {
        int t;

        t = conta;
        conta = t + 1;
    }
}

void main() {
    conta = 0;
    parbegin(incrementa, incrementa);
    write(conta);
}
```

che avvia due processi concorrenti indipendenti i quali incrementano iterativamente una variabile condivisa. Sapendo che la velocità relativa di avanzamento dei due processi non è in alcun modo prevedibile:

1. quali sono i valori minimo e massimo che il risultato può assumere e perché?
2. quali sono i valori minimo e massimo che il risultato può assumere se vengono avviati M processi invece di 2 e perché?

20 - 40

20 - 20M

6.1.7 Chiosco

(punti: 18)

Un piccolo chiosco serve panini con tonno e pomodoro, freddi, e panini con broccoli e salsicce, caldi. I panini vengono preparati nel retro da un garzone, in teglie da 30 panini l'una, e i panini col tonno non possono stare nella stessa teglia dei panini con le salsicce.

I panini vengono serviti sul davanti dalla padrona, che sul banco ha spazio per due sole teglie, una per ripieno. Normalmente la padrona serve i clienti in ordine di arrivo. Quando una delle teglie si esaurisce avverte il garzone, che entro qualche minuto sostituirà la teglia vuota con una piena di panini dello stesso tipo precedente.

Nel frattempo, la padrona va avanti a servire, facendo aspettare temporaneamente quei clienti che chiedono il ripieno mancante. Quando la teglia è stata sostituita, la padrona serve i clienti che precedentemente aveva fatto attendere prima di riprendere il normale servizio in ordine di arrivo. Se mentre il garzone prepara la nuova teglia la padrona esaurisce anche l'altra, dopo aver avvertito il garzone, sospende temporaneamente il servizio.

Scrivere in pseudo-codice i processi garzone, padrona e cliente generico, utilizzando i semafori per la sincronizzazione e tenendo presente che:

- si può trascurare l'avvio delle attività e considerare le due teglie già piene all'avvio;
- è proibito il ricorso all'attesa attiva (*busy waiting*);
- il generico cliente sceglie a caso quale ripieno chiedere;
- il processo garzone ha una priorità alta, per cui quando viene attivato solo una teglia può essere vuota, ma la preparazione della nuova teglia richiede un tempo finito (inferiore rispetto al tempo necessario affinché si presentino 30 clienti), parte del quale lo trascorre in stato di *blocked*;
- quando il garzone sostituisce una teglia sul banco lo fa senza disturbare la padrona (i.e. ordinando correttamente le operazioni non è necessario ricorrere a mutua esclusione).

6.1.8 Incrocio

(punti: 18)

Un incrocio di due grandi viali (uno in direzione Nord-Sud, l'altro in direzione Est-Ovest) è regolato da un vigile. Il vigile lascia passare le macchine su uno dei due viali (in ambedue i sensi di percorrenza) per un tempo T , poi blocca il transito da quel viale, attende che l'incrocio si liberi degli automezzi che lo avevano già impegnato, e quindi dà via libera ai mezzi di trasporto in attesa sull'altro viale, lasciandoli transitare per un tempo T , e così via indefinitamente. I due viali sono così ampi che nell'incrocio non si verifica mai un ingorgo ed il flusso di traffico è sempre scorrevole. Scrivere in pseudo-codice i generici processi "vigile", "automezzo sulla direttrice Nord-Sud", "automezzo sulla direttrice Est-Ovest", utilizzando per la sincronizzazione il meccanismo dei semafori. All'avvio del vigile, l'impegno dell'incrocio dovrà essere già stato assegnato alle macchine in transito sulla direttrice Nord-Sud.

6.1.9 Buffer

(punti: 18)

Un processo P produce informazioni che accoda in un *buffer* FIFO b , di capacità massima limitata ad N elementi. Un processo C preleva le informazioni dal *buffer* b per farne qualcosa di utile. Un processo S , ogni tanto, seleziona a caso due indici i e j nel *buffer* b e scambia tra loro le informazioni relative.

I processi accedono al *buffer* b tramite una libreria già fornita, che implementa le seguenti funzioni: `put(b, x)` e `get(b)` rispettivamente aggiungono un elemento in coda e lo prelevano dalla testa; `write(b, k, x)` e `read(b, k)` rispettivamente consentono di sostituire o leggere l'elemento alla posizione k -esima, senza cambiare il numero di elementi nel *buffer* (l'elemento in testa ha indice 0). Per un errore di progettazione della libreria, non è possibile sapere quanti elementi sono presenti nel *buffer* ad un dato istante. Per di più, la libreria non è stata progettata per applicazioni concorrenti ed accessi contemporanei al *buffer* da parte di più processi hanno esito imprevedibile.

Implementare in pseudo-codice i processi P , C ed S , utilizzando i semafori per la sincronizzazione. Per selezionare casualmente gli indici in S , utilizzare la funzione `randint(n)` che riporta un intero pseudocasuale, maggiore o uguale a 0 e minore dell'argomento n .

6.1.10 Implementazione di un semaforo generico e di un semaforo binario

(punti: 16)

Un particolare sistema di supporto alla mutua esclusione utilizza delle variabili di tipo `lock_t` e tre chiamate.

`init_lock(lock_t *l)` inizializza la variabile di *lock* allo stato libero, e deve essere obbligatoriamente chiamata prima di poter utilizzare la variabile in questione.

`get_lock(lock_t *l)` consente al processo chiamante di acquisire il *lock* e proseguire, se il *lock* era libero, altrimenti blocca il processo su una coda FIFO associata alla variabile di *lock*.

`release_lock(lock_t *l)` sblocca il primo processo in coda, se presente, altrimenti riporta il *lock* allo stato libero.

Attenzione: `release_lock()` è comunque efficace, anche se il processo chiamante non è in possesso del *lock*.

Utilizzando il sistema descritto, implementare:

- un sistema di semaforo generico, scrivendo il codice per la struttura dati `sem_t` e le routine `init(sem_t *l, int n)`, `wait(sem_t *l)` e `signal(sem_t *l)`;
- un sistema di semaforo binario, scrivendo il codice per la struttura dati `bsem_t` e le routine `binit(bsem_t *l, bool b)`, `bwait(bsem_t *l)` e `bsignal(bsem_t *l)`.

6.1.11 Ponte

(punti: 16)

Un torrente è attraversato da un ponte. Il ponte è lungo quanto tre macchine, ma è largo come una macchina, per cui la circolazione avviene a senso unico alternato. Le regole di circolazione sono:

1. indipendentemente dalla riva da cui provengono, le macchine impegnano il ponte nell'ordine di arrivo;
2. una macchina non può impegnare il ponte finché questo è occupato da una o più macchine provenienti dalla riva opposta;
3. una macchina non può impegnare il ponte se questo è già occupato da tre macchine.

Scrivere in pseudocodice i programmi corrispondenti a macchine provenienti dalla riva destra e dalla riva sinistra, utilizzando i semafori per la sincronizzazione.

6.1.12 Filari

(punti: 20)

Fausto e Piero devono potare una vigna di 20 filari. Filare per filare, iniziando sempre dallo stesso lato, Fausto sceglie su ogni pianta i due tralci da mantenere, tagliando il resto. Piero lo segue, rimuovendo dall'impalcatura del filare quanto Fausto ha tagliato. Il lavoro di Piero richiede più tempo, per cui Fausto, arrivato alla fine di ogni filare, torna indietro ad aiutarlo. Per ragioni di sicurezza, non si può lavorare in due contemporaneamente sulla stessa pianta. Ogni filare ha un numero diverso di piante, e Piero e Fausto non lo conoscono in anticipo. Scrivere in pseudocodice i processi generici corrispondenti a Fausto e Piero, utilizzando i semafori per la sincronizzazione e la funzione *ultima(i, j)*, che riporta 1 se la pianta i-esima è l'ultima del filare j-esimo, 0 se non lo è, un valore casuale se la coppia (i, j) non corrisponde ad una pianta realmente esistente.

6.1.13 Filari con stallo

(punti: 20)

Con riferimento al testo dell'esercizio precedente, di seguito viene proposta una particolare soluzione in cui Fausto, quando ha finito di tagliare il filare corrente, anzichè ripartire dall'ultimo tralcio non ancora raccolto da Piero, parte dall'ultimo tralcio della fila (ovvero quello che ha appena tagliato) e indietreggia.

Benchè sia possibile impostare la soluzione dell'esercizio dei filari secondo questa strategia, il codice che segue *non è corretto*, in quanto può generare una situazione di stallo.

```
#define FILARI 20

shared int n = 0;
shared int filare = 0;
sem piante = 0;
sem mutex = 1;

fausto() {
    int i, mn;

    do {
        i = -1;
        do {
            ++i;
            taglia(i, filare);
            signal(piante);
        } while(!ultima(i, filare));

        ++i;
        do {
            --i;
            wait(mutex);
            if (i < n) {          /* Piero ha preso in carico l'ultima pianta? */
                signal(mutex);
                break;
            }
            mn = n;
            wait(piante);
            signal(mutex);
            rimuovitralci(i, filare);
        } while(i != mn);      /* esco se ho lavorato l'ultima pianta */
        ++filare;
        n = 0;
    } while (filare < FILARI);
    signal(piante);          /* sblocco Piero alla fine del lavoro */
}

piero() {
    int i, j;
```

```

while (1) {
    wait(mutex);
    wait(piante);
    i = n;
    j = filare;
    ++n;
    signal(mutex);
    if (j == FILARI)
        break;
    rimuovitranci(i, j);
}
}

```

Si richiede di fornire una sequenza di esecuzione dei processi fausto() e piero() che genera una situazione di stallo.

6.1.14 Fast Food

(punti: 20)

In un fast food Sam frigge le patatine, in lotti da 20 porzioni, e le mette nel banco riscaldato. Ogni volta che un cliente le richiede, Eric preleva una porzione dal banco. Quando preleva l'ultima, Eric chiede a Sam di produrre un altro lotto. La responsabile dell'esercizio, Judy, veglia sulla qualità del cibo, ed esige che le patatine che sono state per più di 20 minuti nel banco riscaldato vengano buttate e sostituite da altre appena fritte, anche a costo di sospendere il servizio. Scrivere in pseudocodice i processi generici corrispondenti a Sam, Eric e Judy. Utilizzare i semafori per la sincronizzazione. Trascurare i dettagli dell'interazione tra Eric ed ogni cliente. Per attendere il momento di intervenire, Judy utilizza la funzione *waittimer()*, che la sospende finché non scatta un allarme a tempo. La funzione *settimer(n)* fa sì che il temporizzatore scatti dopo *n* minuti, dimenticando qualunque allarme precedentemente impostato.

6.2 Soluzioni

6.2.1 Semafori

Risposte corrette: (b), (c), (f), (g).

6.2.2 Mutua esclusione

Risposte corrette: (a), (d).

6.2.3 Sezione critica

Risposte corrette: (c), (d).

Una *sezione critica* è una sezione di codice di un processo che richiede accesso a risorse condivise e che quindi non può essere eseguita mentre un'altro processo è in esecuzione di una corrispondente sezione critica. Ci si trova così in una situazione di *race (condition)*, in cui se due sezioni *critiche* vengono eseguite "contemporaneamente" il risultato dell'esecuzione complessiva dipende dall'ordine di esecuzione relativo delle singole istruzioni nei due processi. Dalla definizione segue direttamente la veridicità delle affermazioni (c) e (d).

Non essendo la *sezione critica* una risorsa ma solo un pezzo di codice, questo esclude le risposte (a) e (b). Inoltre, anche se per motivi di efficienza è bene che la sezione critica contenga meno istruzioni possibili non ci sono limiti nel numero minimo o massimo di istruzioni che può/deve contenere. Questo esclude le risposte (g), (h), (i), (j) e (k). L'esecuzione della sezione critica, anche se influenza il tempo complessivo di esecuzione, influenza la velocità di esecuzione sono del corrispondente pezzo di codice. E' quindi falsa l'affermazione (e). Infine, la sezione critica non deve essere eseguita necessariamente, questo esclude anche la risposta (f).

6.2.4 Problema lettori-scrittori

Risposte corrette: (a), (b), (c).

6.2.5 Comunicazione tra processi tramite scambio di messaggi

Lo scambio di messaggi consente il passaggio di informazioni e la sincronizzazione tra processi differenti, tramite l'uso delle primitive `send(destinatario, messaggio)` e `receive(mittente, messaggio)`. Spedizione e ricezione possono essere bloccanti o non bloccanti. Destinatario e mittente possono essere specificati direttamente, usando un identificatore di processo, o indirettamente, tramite mailbox. L'indirizzamento indiretto è più flessibile, consentendo di realizzare in modo semplice anche forme di comunicazione "uno a molti" e "molti a molti".

6.2.6 Race condition

Il codice ha una race-condition potenziale nei due statement:

```
t = conta;
conta = t + 1;
```

che porta a risultati diversi a seconda di come i processi si alternano nell'esecuzione. Se tutti i processi riescono ognuno ad eseguire i due statement senza interruzioni, la variabile `conta` subirà gli incrementi effettuati da tutti i processi avviati. Se invece un processo viene interrotto subito dopo il primo dei due statement, quando riprenderà ed eseguirà il secondo vanificherà l'effetto di tutti gli incrementi operati contemporaneamente dagli altri. Il risultato è imprevedibile a priori, ma il minimo corrisponderà ad una situazione di interruzione ad ogni iterazione, e sarà ovviamente pari a 20; il massimo corrisponderà alla situazione nella quale tutti gli incrementi operati vanno a buon fine, e sarà pari a $20 \times M$.

6.2.7 Chiosco

L'esercizio combina due problemi produttore-consumatore, uno tra il garzone e la padrona, l'altro tra la padrona e i clienti. Il secondo è un po' diverso dal problema base, in quanto ogni consumatore consuma un solo elemento, vi sono due tipi diversi di elementi, i consumatori vengono divisi in due classi (e su code diverse).

Uno degli approcci possibili è il seguente:

```
#define PANINI 30

variabili condivise:

int panino = 0; /* 1 -> tonno e pomodoro; 2 -> salsiccia e broccoli */
int esaurito = 0;
int ninattesa = 0;
int n[3] = {0, 0, 0}; /* numero di panini nella teglia */

sem nuovateglia = 0;
sem servizio = 1;
sem coda = 0;
sem richiesta = 0;
sem inattesa = 0;
sem servito = 0;
sem soldi = 0;

garzone() {
    int e;

    while(1) {
        wait(nuovateglia);
        e = esaurito;
        scambiateglie(e, produci(e));
        n[e] = PANINI;
        signal(servizio);
    }
}

cliente() {
    int p, q;
```

```

p = panino_random(); /* 1 o 2 */
wait(coda);
panino = p;
q = esaurito;

if (p == q) ++ninattesa; /* va qui per evitare race
                           condition con la padrona */

signal(richiesta);

if (p == q) wait(inattesa);

wait(servito);
prende(p);
paga(p);
signal(soldi);
mangia(p);
}

serve(int p) { /* evita duplicazioni di codice */
    consegna(p);
    --n[p];
    signal(servito);
    wait(soldi);
    riscuote(p);
}

padrona() {
    int manca = 0;

    while(1) {
        signal(coda);
        wait(richiesta);
        if (panino != esaurito) {
            serve(panino);
            if (n[panino] == 0) {
                esaurito = panino;
                signal(nuovateglia);
                wait(servizio);
            }
        }
        if (n[manca] > 0) {
            while(ninattesa--) {
                signal(inattesa);
                serve(manca);
            }
            if (manca == esaurito) esaurito = 0;
        }
        manca = esaurito;
    }
}

```

In questo approccio, il cliente collabora esplicitamente nel caso che chieda un panino esaurito. Il semaforo servizio consente alla padrona di continuare a servire i clienti quando una teglia è esaurita, ma la blocca qualora si esaurisse anche la seconda.

E' possibile ottenere una versione un po' più "compatta" se i clienti si accodano comunque, dopo aver fatto la fila, su due code per le due diverse teglie, lasciando alla padrona il compito di risvegliarli nell'ordine corretto.

```
#define PANINI 30
```


variabili condivise:

```
int panino = 0; /* 1 -> tonno e pomodoro; 2 -> salsiccia e broccoli */
int esaurito = 0;
int ninattesa = 0;
int n[3] = {0, 0, 0}; /* numero di panini nella teglia */
```

```
sem nuovateglia = 0;
sem servizio = 1;
sem coda[3] = {0, 0, 0};
sem richiesta = 0;
sem servito = 0;
sem soldi = 0;
```

```
garzone() {
    int e;

    while(1) {
        wait(nuovateglia);
        e = esaurito;
        scambiateglie(e, produci(e));
        n[e] = PANINI;
        signal(servizio);
    }
}
```

```
cliente() {
    int p, q;

    p = panino_random(); /* 1 o 2 */
    wait(coda[0]);
    panino = p;
    ++ninattesa;
    signal(richiesta);
    wait(coda[p]);
    --ninattesa;
    wait(servito);
    prende(p);
    paga(p);
    signal(soldi);
    mangia(p);
}
```

```
padrona() {
    int manca = 0;

    while(1) {
        if ((n[manca] > 0) && (ninattesa > 0)) {
            panino = manca;
        } else {
            signal(coda[0]);
            wait(richiesta);
        }
        if (panino != esaurito) {
            signal(coda[panino]);
            consegna(panino);
            --n[panino];
            signal(servito);
        }
    }
}
```

```

        wait(soldi)
        riscuote(panino);
        if (n[panino] == 0) {
            esaurito = panino;
            signal(nuovateglia);
            wait(servizio);
        }
    }
    if (n[esaurito] > 0) esaurito = 0;
    if (ninattesa == 0) manca = esaurito;
}
}

```

6.2.8 Incrocio

Il problema si risolve combinando elementi utilizzati nei problemi archetipali di concorrenza. Ecco una possibile soluzione:

```

shared int n = 0;
shared semaphore NS = 1;
shared semaphore EW = 0;
shared semaphore mutexNS = 1;
shared semaphore mutexEW = 1;
shared semaphore incrocio = 1;
shared semaphore conta = 1;

```

```

auto_NS() {
    int mn;

    wait(mutexNS);
    wait(NS);
    signal(mutexNS);

    wait(conta);
    mn = ++n;
    signal(conta);
    if (mn == 1)
        wait(incrocio);
    signal(NS);

    passa_incrocio();

    wait(conta);
    mn = --n;
    signal(conta);
    if (mn == 0)
        signal(incrocio);
}

```

```

auto_EW() {
    int mn;

    wait(mutexEW);
    wait(EW);
    signal(mutexEW);

    wait(conta);
    mn = ++n;
    signal(conta);
}

```

```

    if (mn == 1)
        wait(incrocio);
    signal(EW);

    passa_incrocio();

    wait(conta);
    mn = --n;
    signal(conta);
    if (mn == 0)
        signal(incrocio);
}

vigile() {
    while(1) {
        sleep(T);
        wait(NS);
        wait(incrocio);
        signal(incrocio);
        signal(EW);

        sleep(T);
        wait(EW);
        wait(incrocio);
        signal(incrocio);
        signal(NS);
    }
}

```

Note:

- a) L'inizializzazione assicura che all'inizio abbiano il via le macchine sulla direttrice NS (molti di voi hanno tentato di farlo fare al vigile).
- b) mutexNS e mutexEW fanno sì che una sola macchina per volta si accodi sui semafori NS ed EW, in modo che il vigile possa bloccare al più presto il semaforo allo scadere del tempo, senza dover aspettare molte macchine già in coda (importante curare questo aspetto).
- c) Il semaforo incrocio consente al vigile di aspettare che l'incrocio sia libero prima di dare il via libera sull'altra direzione (questo requisito è importante).

Un'altra soluzione interessante (proposta da alcuni studenti al compito) è la seguente. Due variabili booleane sono utilizzate per dare il via libera alle macchine sulle due direttrici. Se la variabile consente il passaggio, la macchina passa direttamente, altrimenti incrementa un contatore e si accoda su un semaforo. Quando il vigile cambia la direzione di scorrimento, risveglia tante macchine quante risultano dal contatore. Tali macchine ripetono (giustamente!) la procedura già fatta. Questa soluzione ha un difetto, è iniqua perché non rispetta l'ordine di arrivo delle macchine al semaforo: una macchina arrivata a semaforo aperto potrebbe passare prima di una di quelle che si erano dovute bloccare sulla coda. Immaginatevi le proteste ad un incrocio reale!! Tuttavia, in un'applicazione che non ha il requisito forte dell'ordinamento, questa soluzione è molto elegante dal punto di vista delle performance, in quanto le macchine che arrivando trovano il via libera, procedono senza perdere tempo a consultare semafori vari.

6.2.9 Buffer

I processi P e C corrispondono a produttore e consumatore discussi nel problema canonico. L'unica estensione è la necessità di contare gli elementi presenti ad ogni momento nel buffer in modo che il processo S possa operare. Tale conteggio deve essere effettuato atomicamente con l'accodamento ed il prelievo.

```

semaphore mutex = 1;
semaphore pieni = 0;
semaphore vuoti = N;
buffer b;
int inbuffer = 0;

```

```

P() {
    while(1) {
        q = produci();
        wait(vuoti);
        wait(mutex);
        put(b, q);
        ++inbuffer;
        signal(mutex);
        signal(pieni);
    }
}

C() {
    while(1) {
        wait(pieni);
        wait(mutex);
        g = get(b);
        --inbuffer;
        signal(mutex);
        signal(vuoti);
        consuma(g);
    }
}

```

Il processo S è anch'esso molto semplice, l'unica accortezza necessaria per evitare errori di accesso al buffer consiste nell'evitare che si acceda ad elementi che non contengono nulla. A questo fine è necessario basare il calcolo dei due indici casuali sul valore attuale di inbuffer, e proteggere tutta l'operazione con lo stesso mutex adoperato da P e C, in modo che non possa cambiare la situazione a seguito dell'alternanza tra processi.

```

S() {
    while(1) {
        sleep_for_a_while();
        wait(mutex);
        if (inbuffer>1 {
            i = randint(inbuffer);
            j = randint(inbuffer);
            p = read(b, i);
            t = read(b, j);
            write(b, i, t);
            write(b, j, p);
        }
        signal(mutex);
    }
}

```

Note: utilizzare due semafori separati per fare mutua esclusione sul buffer e sul contatore di elementi non è corretto. Anche se questo non provocherebbe errori in P e C, comunque non garantirebbe, in S, la coerenza tra il conteggio usato per estrarre a caso i e j ed il numero di elementi presenti nel buffer al momento dello scambio.

Inserire una wait(pieni) prima che S effettui il mutex sul buffer, oltre a non garantire un numero di elementi sufficienti ad uno scambio, impedirebbe un corretto funzionamento di P e C: alcuni elementi "sfuggirebbero" dal controllo ed inevitabilmente si giungerebbe al blocco del sistema.

6.2.10 Implementazione di un semaforo generico e di un semaforo binario

Semaforo generico

Questa parte è semplice perché il sistema di mutua esclusione fornito può essere utilizzato anche per accodare i processi in attesa al semaforo, SENZA dover introdurre code o altre strutture dati NON fornite dal testo.

Si può realizzare un'implementazione pedissequamente corrispondente alla definizione di semaforo generico. Ecco una possibile soluzione in C:

```

struct mysem {
    ➔ lock_t mutex; /* mutua esclusione sulle modifiche ad e */
    ➔ int e; /* contatore */
    ➔ lock_t queue; /* per l'accodamento */
};

typedef struct my_sem sem_t;

void init(sem_t *l, int n) {
    init_lock(l->mutex);
    e = n;
    init_lock(l->queue);
    get_lock(l->queue); /* impedisce che un processo "passi" quando deve
                           invece bloccarsi */
}

void wait(sem_t *l) {
    int i;

    (get_lock(l->mutex); -
    i = --l->e; -
    (release_lock(l->mutex); -
    if (i<0) get_lock(l->queue); /* si accoda */
}

void signal(sem_t *l) {
    int i;

    (get_lock(l->mutex);
    i = ++l->e; -
    (release_lock(l->mutex);
    if (i<1) release_lock(l->queue); /* sblocca il primo processo in coda */
}

```

Semaforo binario

Questa parte è ancora più semplice se si osserva che il sistema di lock fornito si comporta già, a tutti gli effetti, come un semaforo binario. Quindi è sufficiente, in C:

```

typedef lock_t bsem_t;

void binit(bsem_t *l, bool b) {
    init_lock(l);
    if(!b) get_lock(l);
}

void bwait(bsem_t *l) {
    get_lock(l);
}

void bsignal(bsem_t *l) {
    release_lock(l);
}

```

A questo punto, prima di leggere oltre, andate a rivedere i due tipi di semafori che abbiamo realizzato, simulandone l'esecuzione, in particolare nel caso in cui ci siano più processi che fanno wait() mentre uno fa signal(). C'è una differenza di comportamento, quale? La risposta giusta la trovate più sotto.

La differenza sta nel fatto che mentre il semaforo binario è di tipo "strong", come il sistema di mutua esclusione che abbiamo usato, il semaforo generico è di tipo "weak". Una soluzione "weak" è perfettamente valida, perché l'esercizio non richiede

esplicitamente un semaforo "strong", ma se ci servisse un semaforo "strong" potremmo ottenerlo con semplici modifiche (la signal() rimane immutata):

```
struct mysem {
    lock_t mutex; /* mutua esclusione sulle modifiche ad e */
    lock_t obo;   /* per rendere il semaforo "strong" */
    int e;        /* contatore */
    lock_t queue; /* per l'accodamento */
};

typedef struct my_sem sem_t;

void init(sem_t *l, int n) {
    init_lock(l->mutex);
    init_lock(l->obo);
    e = n;
    init_lock(l->queue);
    get_lock(l->queue); /* impedisce che un processo "passi" quando deve
                        invece bloccarsi */
}

void wait(sem_t *l) {
    int i;

    get_lock(l->obo); /* A (v. sotto) */
    get_lock(l->mutex); /* B (v. sotto) */
    i = --l->e;
    release_lock(l->mutex);
    if (i < 0) get_lock(l->queue); /* si accoda */
    release_lock(l->obo);
}

void signal(sem_t *l) {
    int i;

    get_lock(l->mutex);
    i = ++l->e;
    release_lock(l->mutex);
    if (i < 1) release_lock(l->queue); /* sblocca il primo processo in coda */
}
```

Altra domanda: che succederebbe se nella wait() del semaforo "strong" scambiassimo tra loro le due righe di codice marcate con A e B?

6.2.11 Ponte

Suggerimenti. Per garantire il primo requisito è sufficiente un semaforo utilizzato da tutte le macchine indipendentemente dal verso di provenienza. Lo stesso dicasi per il terzo requisito, che in questo caso sarà inizializzato al valore della portata massima (3).

Verranno fornite 3 diverse soluzioni. Nella seconda e nella terza soluzione verranno introdotti dei vincoli aggiuntivi rispetto alla traccia. Per cui si consiglia di leggere i vincoli di volta in volta suggeriti e di provare a riformulare una soluzione che soddisfi i nuovi requisiti come ulteriore esercizio prima di leggere lo pseudo-codice d'esempio fornito.

Soluzione 1

Una soluzione possibile è costruibile come segue. Innanzitutto la capacità del ponte (requisito (c)) è gestibile con un solo semaforo, opportunamente inizializzato, che protegge la sola fase di transito. In secondo luogo, il requisito (a) richiede la protezione (con mutua esclusione tramite semafori) del protocollo di accesso al ponte, in modo da forzare l'ordinamento di tutte le macchine indipendentemente dalla riva di provenienza. E' lo stesso metodo illustrato dal testo per ordinare i lettori in una delle soluzioni del problema lettori scrittori. Dentro il protocollo di accesso dovrà avvenire il coordinamento relativo al verso di percorrenza ed alle variabili condivise tra macchine provenienti dalla stessa riva. Ecco il codice.

```

semaphore accesso = 1; /* forza l'ordinamento delle macchine che arrivano */
semaphore portata = 3; /* capacita' del ponte */
semaphore mutexsx = 1; /* per controllo di flusso e mutex */
semaphore mutexdx = 1; /* per controllo di flusso e mutex */
int nsx = 0; /* macchine che hanno ottenuto accesso da sx */
int ndx = 0; /* macchine che hanno ottenuto accesso da dx */

```

```

macchina_dalla_riva_sinistra() {
    wait(accesso); /* inizio protocollo di accesso */
    wait(mutexsx); /* per evitare scontri e per mutex su nsx e ndx */
    ++nsx;
    if (nsx==1) /* prima di un gruppo in questa direzione */
        wait(mutexdx); /* blocca macchine dall'altra riva */
    signal(mutexsx);
    signal(accesso); /* fine protocollo di accesso */

    wait(portata);
    passa_ponte();
    signal(portata);

    wait(mutexsx); /* mutex su nsx e ndx */
    --nsx;
    if(nsx==0) /* ultima di un gruppo in questa direzione */
        signal(mutexdx); /* sblocca macchine dall'altra riva */
    signal(mutexsx);
}

```

```

macchina_dalla_riva_destra() {
    wait(accesso); /* inizio protocollo di accesso */
    wait(mutexdx); /* per evitare scontri e per mutex su nsx e ndx */
    ++ndx;
    if (ndx==1) /* prima di un gruppo in questa direzione */
        wait(mutexsx); /* blocca macchine dall'altra riva */
    signal(mutexdx);
    signal(accesso); /* fine protocollo di accesso */

    wait(portata);
    passa_ponte();
    signal(portata);

    wait(mutexdx); /* mutex su nsx e ndx */
    --ndx;
    if(ndx==0) /* ultima di un gruppo in questa direzione */
        signal(mutexsx); /* sblocca macchine dall'altra riva */
    signal(mutexdx);
}

```

Soluzione 2

Di seguito viene proposta una seconda soluzione in cui si usano meno risorse. Prima di procedere con la lettura si invita a provare a risolvere l'esercizio usando una sola variabile intera di conteggio (anziché due) e quattro semafori:

```

int m=0;
sem mutexdx=1;
sem mutexsx=1;
sem portata=3;
sem accesso=1;

```

Il codice di una possibile soluzione è il seguente (suggerito da uno studente).

```
int m=0;
sem mutexdx=1;
sem mutexsx=1;
sem portata=3;
sem accesso=1;

Destra(){

wait(accesso);

wait(mutexdx);
m++;
if (m==1) wait(mutexsx);
    signal(mutexdx);

signal(accesso);

wait(portata);
attraversa_ponte();
signal(portata);

wait(mutexdx);
m--;
if (m==0)
    signal(mutexsx);
signal(mutexdx);

}

Sinistra(){

wait(accesso);

wait(mutexsx);
m++;
if (m==1) wait(mutexdx);
    signal(mutexsx);

signal(accesso);

wait(portata);
attraversa_ponte();
signal(portata);

wait(mutexsx);
m--;
if (m==0)
    signal(mutexdx);
signal(mutexsx);

}
```


Soluzione 3

Proviamo infine altre due variazioni sul tema.

1. Supponiamo che i semafori siano di tipo binario, ossia:

- possono essere inizializzati a 0 o ad 1;
- quando un processo fa wait, se il contatore associato vale 1 lo azzerà e prosegue, altrimenti si blocca nella coda associata;
- quando un processo fa signal, se il contatore è 0 e ci sono processi in coda sveglia il primo, se il contatore è 0 e non ci sono processi in coda porta il contatore ad 1.

Risolvere l'esercizio senza aggiungere altre variabili condivise o semafori.

2. Dimostrare che se si utilizza una sola variabile per contare le macchine che hanno accesso al ponte (chiamiamola m), il numero minimo di semafori necessario per risolvere l'esercizio è 4.

Il codice di una possibile soluzione al punto 1 è il seguente (suggerito da uno studente).

```
int m=0;
sem accesso=1;
sem mutexdx=1;
sem mutexsx=1;
sem portata=1;

Destra(){
int curr;

wait(accesso);

wait(mutexdx);
curr=++m;
if (m==1)
    wait(mutexsx);
signal(mutexdx);

signal(accesso);
if (curr>=3)
    wait(portata);
attraversa_ponte();

wait(mutexdx);
if (m>=3)
    signal(portata);
m--;
if (m==0) signal(mutexsx);
    signal(mutexdx);
}

Sinistra(){
int curr;

wait(accesso);

wait(mutexsx);
curr=++m;
if (m==1)
    wait(mutexdx);
signal(mutexsx);
```

```

signal(accesso);
if (curr>=3)
    wait(portata);
attraversa_ponte();

wait(mutexsx);
if (m>=3)
    signal(portata);
m--;
if (m==0) signal(mutexdx);
    signal(mutexsx);
}

```

Per quanto riguarda il punto 2, la domanda fondamentale a cui è necessario rispondere è: quali attese logicamente indipendenti (ossia dovute a cause diverse) possono essere necessarie ad un'auto? Risposta:

- A. per ordinarsi con le altre in base all'arrivo,
- B. se la sezione critica sulla variabile condivisa m è impegnata,
- C. se il ponte è già impegnato in senso contrario,
- D. se il ponte è troppo pieno.

Totale: 4, quindi 4 semafori. Nel codice proposto per il punto 1, il semaforo *accesso* corrisponde al punto *A*, *portata* corrisponde a *D*, *mutexsx* e *mutexdx* corrispondono alternativamente a *B* e *C* (ma mai *contemporaneamente* alla stessa attesa). Quest'ultimo è un dettaglio implementativo che non inficia la logica sopra descritta.

Una procedura analoga si può utilizzare per altri esercizi. Se in una soluzione ad un esercizio di concorrenza viene utilizzato un numero di semafori superiore a quello minimo necessario, è *probabile* che ci sia qualche problema nell'implementazione. Il ché, tuttavia, non vuol dire che un'implementazione che usa più semafori non possa essere in alcuni casi più conveniente, per semplicità di scrittura o per chiarezza.

6.2.12 Filari

Si tratta di un semplice sistema produttore-consumatore, nel quale ogni tanto il produttore (Fausto) diventa consumatore degli eventi da lui stesso prodotti. Se Fausto non tornasse indietro, l'unica complicazione sarebbe il cambio di filare: il numero delle piante non è noto in anticipo. Tuttavia, il numero di piante su un filare sarà noto non appena Fausto arriverà all'ultima pianta, e quindi potrà essere condiviso in una variabile globale. La temporanea trasformazione di Fausto in consumatore pone un altro problema. Se si utilizzasse un solo semaforo per garantire la rimozione dei tralci solo dalle piante su cui Fausto è già intervenuto, come avviene nel produttore-consumatore standard, si rischierebbe un deadlock al termine del filare. E' anche qui necessario utilizzare il numero di piante che Fausto ha contato nel filare.

Una possibile soluzione è la seguente:

```

#define FILARI 20
#define TROPPEVITI 10000

shared int n = 0;
shared int len = TROPPEVITI;
sem piante = 0;
sem mutex = 1;
sem filaref = 0;
sem filarei = 1;

fausto() {
    int i, j;

    for (j = 0; j < FILARI; ++j) {
        i = 0;
        do {

```

```

        taglia(i, j);
        signal(piante);
        ++i;
    } while(!ultima(i-1,j));
    wait(filarei);
    len = i;

    wait(mutex);
    while((i=n) < len) {
        ++n;
        wait(piante);
        signal(mutex);
        rimuovitranci(i, j);
        wait(mutex);
    }
    signal(mutex);
    signal(filaref);
}

piero() {
    int i, j;

    for (j = 0; j < FILARI; ++j) {
        wait(mutex);
        while((i=n) < len) {
            ++n;
            wait(piante);
            signal(mutex);
            rimuovitranci(i, j);
            wait(mutex);
        }
        signal(mutex);
        wait(filaref);
        n = 0;
        len = TROPPEVITI;
        signal(filarei);
    }
}

```

6.2.13 Filari con stallo

Una sequenza che può portare allo stallo è la seguente.
 Supponiamo la seguente situazione:

- la variabile $n = h$;
- la variabile i , interna a Fausto, è uguale a $k + 1$ ($h < k$).

Fausto è in esecuzione e sta eseguendo il suo ciclo da "consumatore":

- decrementa i (ora $i = k$);
- acquisisce la mutua esclusione;
- pone $mn = n$ (quindi, $mn = h$);
- esegue: `wait(piante)`;
- rilascia la mutua esclusione e inizia la procedura di rimozione del k -esimo tralcio.

A questo punto Fausto viene interrotto e va in esecuzione Piero, che esegue ripetutamente il suo ciclo, di volta in volta incrementando n e decrementando il semaforo piante fino a raggiungere Fausto. Piero agisce sull'ultima pianta rimasta da raccogliere (la $(k-1)$ -esima). Nel far ciò, come al solito, incrementa n (ed ora $n = k$) e decrementa *piante*, che assume il valore 0 (non ci sono più piante da raccogliere). Piero rilascia la mutua esclusione e rimuove la $(k-1)$ -esima pianta, riprendendo quindi il suo ciclo:

- acquisisce nuovamente la mutua esclusione; (!!)
- esegue: *wait(piante)*, cosa che lo porta a bloccarsi, essendo *piante* = 0.

Ritorna in esecuzione Fausto, che finisce di rimuovere il tralcio k -esimo sul quale era stato interrotto. Fausto controlla quindi che mn sia diverso da i . Siccome la variabile mn è locale a Fausto (!!), essa conserva il vecchio valore, acquisito nell'iterazione precedente; e, infatti, abbiamo che $mn = h$. Per cui la condizione è soddisfatta e Fausto continua il suo ciclo:

- decrementa i ;
- esegue: *wait(mutex)* e... si blocca, perchè la mutua esclusione non è stata rilasciata da Piero prima che questi si bloccasse a sua volta su *piante*.

6.2.14 Fast Food

L'esercizio è un'estensione del problema produttore consumatore, con due consumatori che seguono modi di consumo differenti. La produzione ed il consumo sono semplificati dal fatto che gli oggetti (i lotti) sono prodotti e consumati uno per volta. L'unico aspetto delicato è la gestione del timeout di Judy, che deve scadere 20 minuti dopo l'ultima frittura. Se le porzioni "vanno via" rapidamente, ed Eric chiede a Sam di friggere un nuovo lotto, se Judy si attiva tra la richiesta di Eric e il momento in cui il timer viene aggiornato, c'è il rischio concreto che le patatine vengano buttate appena fritte. Per risolvere questo problema è necessario ipotizzare esplicitamente che Judy abbia una priorità più alta degli altri due processi. Il che è suggerito dal testo quando informa che Judy è la responsabile dell'esercizio.

Vediamo una possibile soluzione:

```
#define PORZIONI 20
#define TIMEOUT 20

shared int porzioni = 0;
sem cliente = 0;
sem produci = 1;
sem pronte = 0;
sem x = 1;

sam() {
    while (1) {
        wait(produci);
        if (porzioni) {
            butta_patatine();
        }
        frigge_patatine();
        le_mette_nel_banco();
        settimer(TIMEOUT);
        porzioni = PORZIONI;
        signal(pronte);
    }
}

eric() {
    int p;

    wait(pronte);
    while(1) {
        wait(cliente);
        wait(x);
        --porzioni;
        p = porzioni;
```

```

    prende_porzione();
    signal(x);
    servi_cliente();
    if (!p) {
        signal(produci);
        wait(pronte);
    }
}
}

judy() {
/* ogni volta che si risveglia Judy causa la preemption di Sam ed Eric */

    while (1) {
        waittimer();
        wait(x);
        if (porzioni) {
            signal(produci);
            wait(pronte);
        }
        signal(x);
    }
}

```

Note.

- a) Judy fa buttare patatine fritte da meno di 20 minuti. E' necessario esplicitare l'assunzione sulla priorità.
 - b) L'esercizio richiede di trascurare i dettagli dell'interazione col cliente. Ciò significa che non è necessario prevedere azioni del tipo: il cliente paga, Eric prende i soldi e dà una ricevuta, ecc. Tuttavia Eric deve attendere un cliente prima di servirlo. Questo per soddisfare il requisito: "Ogni volta che un cliente le richiede, Eric preleva una porzione dal banco".
 - c) Non è necessario un ulteriore semaforo per forzare la mutua esclusione tra Sam ed Eric sul banco: la mutua esclusione è già garantita da produzione/consumo un lotto per volta.
 - d) Le specifiche richiedono che Judy intervenga a 20 minuti dall'ultima frittura, non ogni 20 minuti indipendentemente da ciò che fa Sam.
 - e) Le specifiche richiedono che sia Sam ad attivare Eric quando ve n'è bisogno. Viceversa, una soluzione in cui si fa friggere Sam a ciclo continuo è irragionevole.
 - f) Secondo le specifiche, Eric chiede nuove patatine quando le finisce, senza aspettare che arrivi un nuovo cliente. Quest'ultimo approccio (aspettare che arrivi un nuovo cliente) è sicuramente superiore per l'ambiente e l'economia ma non è quanto richiesto dalle specifiche, e non è accettabile come soluzione. E' sempre necessario seguire le specifiche della traccia, anche quando, come in questo caso, si richiede un protocollo di comportamento degli attori coinvolti poco realistico (o poco economico o poco ambientalista...).
 - g) Un'altra alterazione delle specifiche non accettabile è far sì che Judy non parli con Sam, ma lasci il compito ad Eric. Questo non è conveniente, perché le patate non vengono sostituite immediatamente, ma solo quando arriva un cliente, come al punto precedente. In genere questa violazione delle specifiche va in coppia con un'altra: Judy butta personalmente le patatine. Tuttavia, quest'ultima violazione (da sola) non è di sostanza e può essere accettata.
- Infine, un'altra violazione delle specifiche, anche questa non di sostanza è la seguente: Sam non si rivolge ad Eric direttamente, ma lo fa tramite Judy, facendo scattare immediatamente il timer. Questo elimina alla radice, in modo assai elegante, il problema dell'attivazione di Judy mentre Sam frigge. Eccone un'implementazione:

```

#define PORZIONI 20
#define TIMEOUT 20

shared int porzioni = 0;
sem cliente = 0;
sem produci = 1;
sem pronte = 0;

sam() {
    while (1) {
        wait(produci);
        if (porzioni) {

```

```

        butta_patatine();
    }
    frigge_patatine();
    le_mette_nel_banco();
    settimer(TIMEOUT);
    porzioni = PORZIONI;
    signal(pronte);
}
}

eric() {
    while(1) {
        wait(cliente);
        wait(pronte);
        --porzioni;
        prende_porzione();
        if (!porzioni)
            settimer(0);
        signal(pronte);
        serve_cliente();
    }
}

judy() {
/* ogni volta che si risveglia Judy causa la preemption di Sam ed Eric */

    while (1) {
        waittimer();
        wait(pronte);
        signal(produci);
    }
}

```

Chapter 7

Stallo^{NE}

7.1 Testi

7.1.1 Attesa indefinita

(punti: -1,4)

Riguardo l'attesa indefinita (*starvation*), si può dire che:

- (a) è un caso particolare dello stallo; lo stallo è il deadlock
- ~~(b) è causata dall'algoritmo del banchiere per differire l'esecuzione di processi che potrebbero portare il sistema in uno stato non sicuro;~~
- (c) è intenzionalmente provocata da alcuni *scheduler* per differire l'esecuzione di processi che potrebbero portare il sistema in stallo;
- (d) è intenzionalmente provocata da alcuni *scheduler* per differire l'esecuzione di processi di minore priorità;
- (e) è intenzionalmente provocata da alcuni *scheduler* per differire l'esecuzione di processi di lunga durata;
- (f) può essere risolta con tecniche di *detection and removal*;
- (g) può essere prevenuta con tecniche di **programmazione** opportune;
- (h) può essere evitata impedendo ai processi di andare nello stato *blocked*;
- ~~(i) nessuna delle affermazioni precedenti è corretta;~~

7.1.2 Prevenzione, esclusione e rimozione dell'attesa circolare

(punti: -1,4)

Il problema dello stallo (*deadlock*) è generalmente affrontato intervenendo sulla condizione di attesa circolare con tre possibili approcci: prevenzione dell'attesa circolare, esclusione dell'attesa circolare, identificazione e rimozione dell'attesa circolare (*prevention, avoidance, detection and removal*). Quali delle seguenti affermazioni sono corrette?

- (a) l'identificazione e rimozione compete al programmatore delle applicazioni;
- (b) l'identificazione e rimozione è effettuata dal sistema operativo ad ogni richiesta di risorsa;
- (c) l'identificazione e rimozione è effettuata periodicamente dall'amministratore di sistema;
- (d) l'identificazione e rimozione garantisce che non si verificherà mai stallo;
- (e) l'esclusione compete al programmatore delle applicazioni;
- (f) l'esclusione è effettuata dal sistema operativo ad ogni richiesta di risorsa;
- (g) l'esclusione è effettuata periodicamente dall'amministratore di sistema;
- (h) l'esclusione garantisce che non si verificherà mai stallo;
- (i) la prevenzione compete al programmatore delle applicazioni;
- (l) la prevenzione è effettuata dal sistema operativo ad ogni richiesta di risorsa;
- (m) la prevenzione è effettuata periodicamente dall'amministratore di sistema;
- (n) la prevenzione garantisce che non si verificherà mai stallo.

7.1.3 Condizioni dello stallo

(punti: 7)

Illustrare in al più 70 parole le condizioni dello stallo (*deadlock*).

7.2 Soluzioni

7.2.1 Attesa indefinita

Risposte corrette: (i).

7.2.2 Prevenzione, esclusione e rimozione dell'attesa circolare

Risposte corrette: (f), (h), (i).

7.2.3 Condizioni dello stallo

E' sufficiente elencare le quattro condizioni (mutua esclusione, assenza di prerilascio, hold & wait, attesa circolare), specificando che ognuna è necessaria e che tutte insieme sono sufficienti. Attenzione a distinguere i concetti di necessità e di sufficienza di una condizione.