Seconda parte

Gestione dei processi

Un *processo* si può pensare come un programma in esecuzione. Per svolgere il proprio compito, un processo richiede determinate risorse, come tempo d'elaborazione della CPU, memoria, file e dispositivi di I/O. Queste risorse si assegnano al processo al momento della sua creazione o durante l'esecuzione.

Il processo è l'unità di lavoro nella maggior parte dei sistemi. Un sistema tipico è formato da processi del sistema operativo, che eseguono il codice di sistema, e da processi utenti, che eseguono il codice utente. Tutti questi processi sono eseguibili concorrentemente.

Benché tradizionalmente un processo contenga un solo thread di controllo dell'esecuzione, la maggior parte dei sistemi operativi moderni attualmente gestisce processi con più thread.

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei processi e dei thread di sistema e utenti: creazione e cancellazione, scheduling, offerta dei meccanismi di sincronizzazione e comunicazione, gestione delle situazioni di stallo.

Capitolo 3

Processi

OBIETTIVI

- Introduzione del concetto di processo – un programma in esecuzione – che forma la base della computazione.
- Spiegazione delle diverse caratteristiche legate ai processi, comprese quelle relative a scheduling, creazione e terminazione, e comunicazione tra processi.
- Descrizione della comunicazione nei sistemi client-server.

I primi sistemi di calcolo consentivano l'esecuzione di un solo programma alla volta, che aveva il completo controllo del sistema e accesso a tutte le sue risorse. Gli attuali sistemi di calcolo consentono, invece, che più programmi siano caricati in memoria ed eseguiti in modo concorrente. Tale evoluzione richiede un più severo controllo e una maggiore compartimentazione dei vari programmi. Da tali necessità deriva la nozione di **processo d'elaborazione** – o, più brevemente, **processo** – che in prima istanza si può definire come un programma in esecuzione, e costituisce l'unità di lavoro dei moderni sistemi a partizione del tempo d'elaborazione.

Maggiore è la complessità di un sistema operativo, maggiori sono i servizi che si suppone esso fornisca ai propri utenti. Benché il suo compito principale sia l'esecuzione dei programmi utenti, deve anche occuparsi dei vari compiti di sistema che è più conveniente lasciare fuori dal kernel. Un sistema è quindi costituito da un insieme di processi: quelli del sistema operativo eseguono il codice di sistema; gli utenti il codice utente. Tutti questi processi si possono eseguire potenzialmente in modo concorrente e l'uso della CPU (o di più unità d'elaborazione) è commutato tra i vari processi. Il sistema operativo può rendere il calcolatore più produttivo avvicendando i diversi processi nell'uso della CPU. In questo capitolo parleremo dei processi e del loro funzionamento.

3.1 Concetto di processo

Una questione che sorge dall'analisi dei sistemi operativi è la definizione delle attività della CPU. Un **sistema a lotti** (*batch*) esegue **lavori** (*job*), mentre un sistema a partizione del tempo esegue **programmi utenti** o **task**. Persino in un sistema monoutente, come Microsoft Windows, un utente può far eseguire diversi programmi contemporaneamente: un elaboratore di testi, un programma di consultazione del Web, un programma per la posta elettroni-

ca. Anche se l'utente esegue un solo programma alla volta, il sistema operativo deve svolgere le proprie attività interne, per esempio la gestione della memoria. Queste attività sono simili per molti aspetti, perciò sono denominate *processi*.

In questo testo i termini *lavoro* e *processo* sono usati in modo quasi intercambiabile. Sebbene si preferisca il termine *processo*, occorre ricordare che la maggior parte della terminologia e della teoria dei sistemi operativi si è sviluppata in un periodo in cui l'attività principale dei sistemi operativi riguardava la gestione dei lavori d'elaborazione in sistemi a lotti. Sarebbe fuorviante evitare di usare termini comunemente accettati che contengono la parola *lavoro* o *job*, per esempio *job scheduling*, solo perché il termine *processo* ha ormai soppiantato il termine *lavoro*.

3.1.1 Processo

Informalmente, un *processo* è un programma in esecuzione. È qualcosa di più del codice di un programma, talvolta noto anche come **sezione di testo**: comprende l'attività corrente, rappresentata dal valore del **contatore di programma** e dal contenuto dei registri della CPU; normalmente comprende anche la propria **pila** (*stack*), contenente a sua volta i dati temporanei, come i parametri di un metodo, gli indirizzi di rientro e le variabili locali, e una **sezione di dati** contenente le variabili globali. Un processo può includere uno **heap**, ossia della memoria dinamicamente allocata durante l'esecuzione del processo. La struttura di un processo in memoria è illustrata nella Figura 3.1.

Sottolineiamo che un programma di per sé non è un processo; un programma è un'entità *passiva*, come il contenuto di un file memorizzato in un disco, mentre un processo è un'entità *attiva*, con un contatore di programma che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo allorquando il file eseguibile che lo contiene è caricato in memoria. Due tecniche comuni per ottenere questo effetto sono il doppio clic sull'icona del file seguibile e la digitazione del nome del file eseguibile nella riga di comando. (Esempi tipici di nomi di file eseguibili: `prog.exe` oppure `a.out`.)

Sebbene due processi siano associabili allo stesso programma, sono tuttavia da considerare due sequenze d'esecuzione distinte. Alcuni utenti possono, per esempio, far eseguire diverse istanze dello stesso programma di posta elettronica, così come un utente può invocare più istanze dello stesso programma di consultazione del Web. Ciascuna di loro è un diverso processo, e benché le sezioni di testo siano equivalenti, quelle dei dati, dello heap e

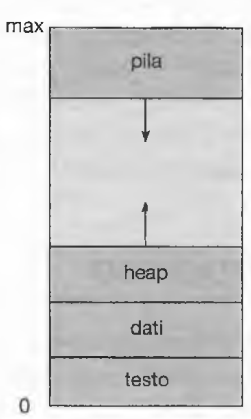

Figura 3.1 Processo in memoria.

della pila sono diverse. È inoltre usuale che durante la propria esecuzione un processo generi altri processi. Questo argomento è trattato nel Paragrafo 3.4.

3.1.2 Stato del processo

Un processo durante l'esecuzione è soggetto a cambiamenti di **stato**, definiti in parte dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati.

- **Nuovo.** Si crea il processo.
- **Esecuzione.** Un'unità d'elaborazione esegue le istruzioni del relativo programma.
- **Attesa.** Il processo attende che si verifichi qualche evento (come il completamento di un'operazione di I/O o la ricezione di un segnale).
- **Pronto.** Il processo attende di essere assegnato a un'unità d'elaborazione.
- **Terminato.** Il processo ha terminato l'esecuzione.

Queste definizioni sono piuttosto arbitrarie, e variano secondo il sistema operativo. Gli stati che rappresentano sono in ogni modo presenti in tutti i sistemi, anche se alcuni sistemi operativi introducono ulteriori distinzioni tra gli stati dei processi. In ciascuna unità d'elaborazione può essere in *esecuzione* solo un processo per volta, sebbene molti processi possano essere *pronti* o nello stato di *attesa*. Il diagramma di transizione corrispondente a questi stati è riportato nella Figura 3.2.

3.1.3 Blocco di controllo dei processi

Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo di un processo** (*process control block*, PCB, o *task control block*, TCB). Un blocco di controllo di un processo (Figura 3.3) contiene molte informazioni connesse a un processo specifico, tra cui le seguenti.

- **Stato del processo.** Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.
- **Contatore di programma.** Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.

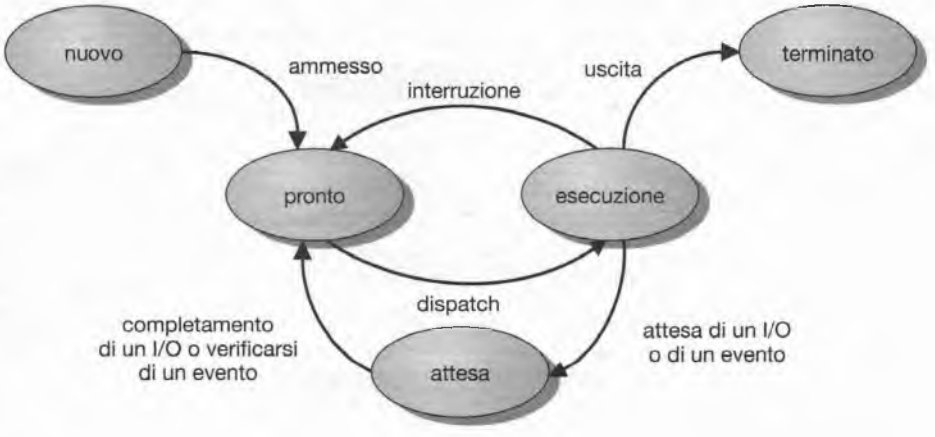

Figura 3.2 Diagramma di transizione degli stati di un processo.

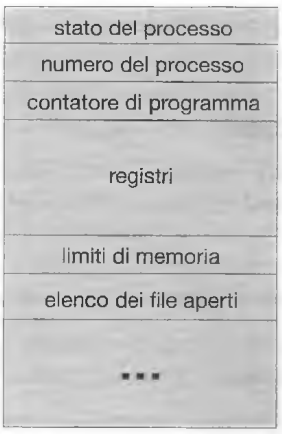

Figura 3.3 Blocco di controllo di un processo (PCB).

- **Registri di CPU.** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri d'indice, puntatori alla cima delle strutture a pila (*stack pointer*), registri d'uso generale e registri contenenti informazioni relative ai codici di condizione. Quando si verifica un'interruzione della CPU, si devono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo (Figura 3.4).
- **Informazioni sullo scheduling di CPU.** Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling (nel Capitolo 5 si descrive lo scheduling dei processi).

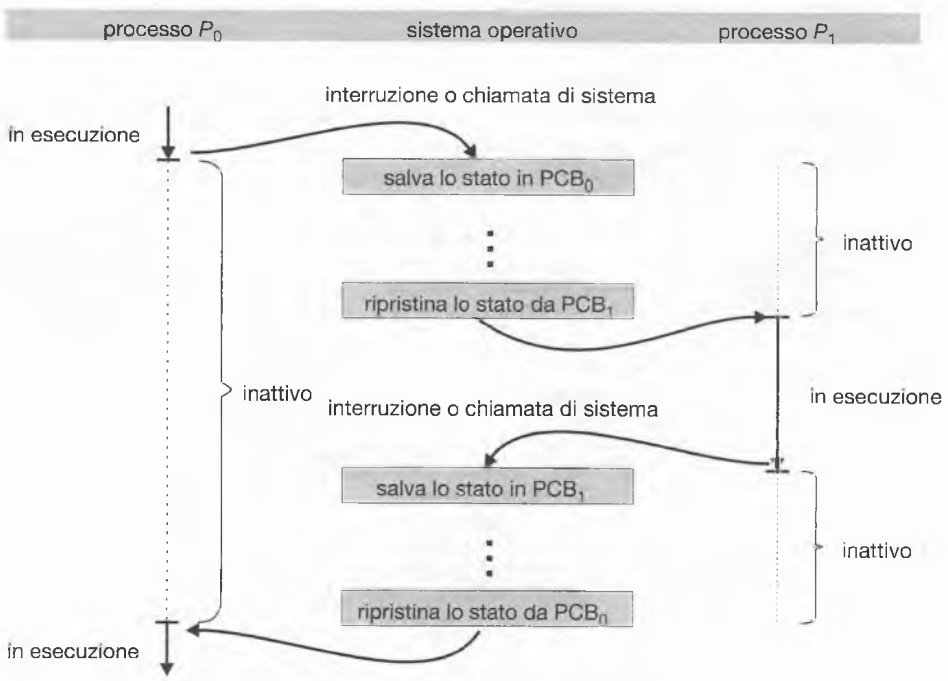

Figura 3.4 La CPU può essere commutata tra i processi.

- **Informazioni sulla gestione della memoria.** Queste informazioni si possono esprimere attraverso i valori dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo (Capitolo 8).
- **Informazioni di contabilizzazione delle risorse.** Queste informazioni comprendono il tempo d'uso della CPU e il tempo reale d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.
- **Informazioni sullo stato dell'I/O.** Queste informazioni comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti, e così via.

In sintesi, il PCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

3.1.4 Thread

Il modello dei processi illustrato fin qui sottintende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione, detto **thread**. Se un processo sta, per esempio, eseguendo un programma di elaborazione di testi, l'esecuzione avviene seguendo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta. Tramite lo stesso processo, per esempio, un utente non può contemporaneamente inserire caratteri e verificare la correttezza ortografica di quel che sta scrivendo. In molti sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più percorsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta. Il Capitolo 4 è dedicato ai processi multithread.

3.2 Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della CPU. L'obiettivo della partizione del tempo è di commutare l'uso della CPU tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo **scheduler dei processi** seleziona un processo da eseguire dall'insieme di quelli disponibili. Nei sistemi monoprocessore non vi sarà mai più di un processo in esecuzione: gli altri dovranno attendere finché la CPU sia nuovamente disponibile.

3.2.1 Code di scheduling

Ogni processo è inserito in una **coda di processi**, composta da tutti i processi del sistema. I processi presenti in memoria centrale, che sono pronti e nell'attesa d'essere eseguiti, si trovano in una lista detta **coda dei processi pronti** (*ready queue*). Questa coda generalmente si memorizza come una lista concatenata: un'intestazione della coda dei processi pronti contiene i puntatori al primo e all'ultimo PCB dell'elenco, e ciascun PCB è esteso con un campo puntatore che indica il successivo processo contenuto nella coda dei processi pronti.

Il sistema operativo ha anche altre code. Quando si assegna la CPU a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta di I/O. Una richiesta di I/O può essere diretta a un dispositivo condiviso, come un disco. Poiché nel sistema esistono molti processi, il disco può essere occupato con una richiesta di I/O di qualche altro processo, quindi il processo deve attendere che il disco sia disponibile.

RAPPRESENTAZIONE DEI PROCESSI DI LINUX

Il blocco di controllo dei processi nel sistema operativo Linux è rappresentato dalla struttura C `task_struct`, contenente una descrizione completa del processo, compreso il suo stato, informazioni sullo scheduling e sulla gestione della memoria, la lista dei file aperti, e puntatori al processo padre e agli eventuali figli. (Il *padre* o *genitore* di un processo è il processo che lo ha creato; i *figli* sono i processi generati.) Alcuni dei campi sono i seguenti:

```
pid_t pid; /* identificatore del processo */
long state; /* stato del processo */
unsigned int time_slice /* informazioni per lo scheduling */
struct files_struct *files; /* lista dei file aperti */
struct mm_struct *mm; /* spazio degli indirizzi del processo */
```

Per esempio, lo stato del processo è rappresentato dal campo `long state`. In Linux l'insieme dei processi è rappresentato da una lista doppia (in cui, cioè, ogni elemento punta al predecessore e al successore) di `task_struct`, e il kernel mantiene un puntatore di nome `current` al processo attualmente in esecuzione. Si veda la Figura 3.5.

Per illustrare le manipolazioni eseguite dal kernel sui campi della struttura, supponiamo che il sistema debba cambiare lo stato del processo in esecuzione al valore `nuovo_stato`; se `current` punta, come detto poc'anzi, al processo attualmente in esecuzione, l'istruzione da eseguire è:

```
current->state = nuovo_stato;
```

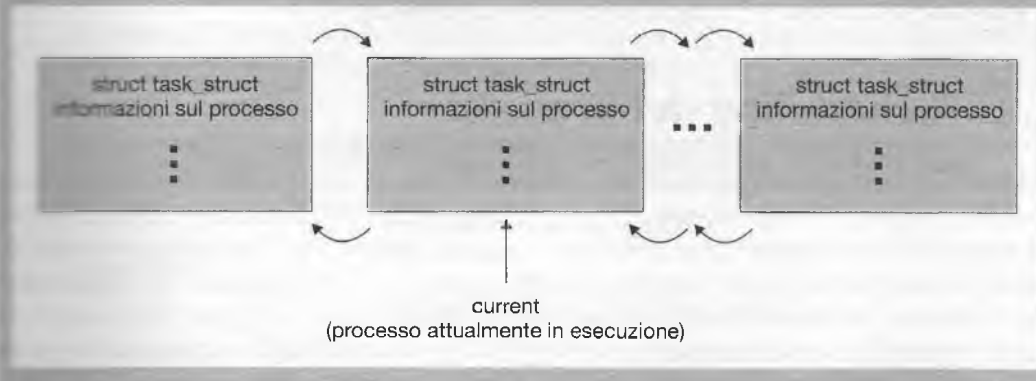

Figura 3.5 Processi attivi di Linux.

L'elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama coda del **dispositivo**; ogni dispositivo ha la propria coda (Figura 3.6).

Una comune rappresentazione utile alla descrizione dello scheduling dei processi è data da un **diagramma di accodamento** come quello illustrato nella Figura 3.7. Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la coda dei processi pronti e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

Un nuovo processo si colloca inizialmente nella coda dei processi pronti, dove attende finché non è selezionato per essere eseguito (*dispatched*). Una volta che il processo è assegnato alla CPU ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

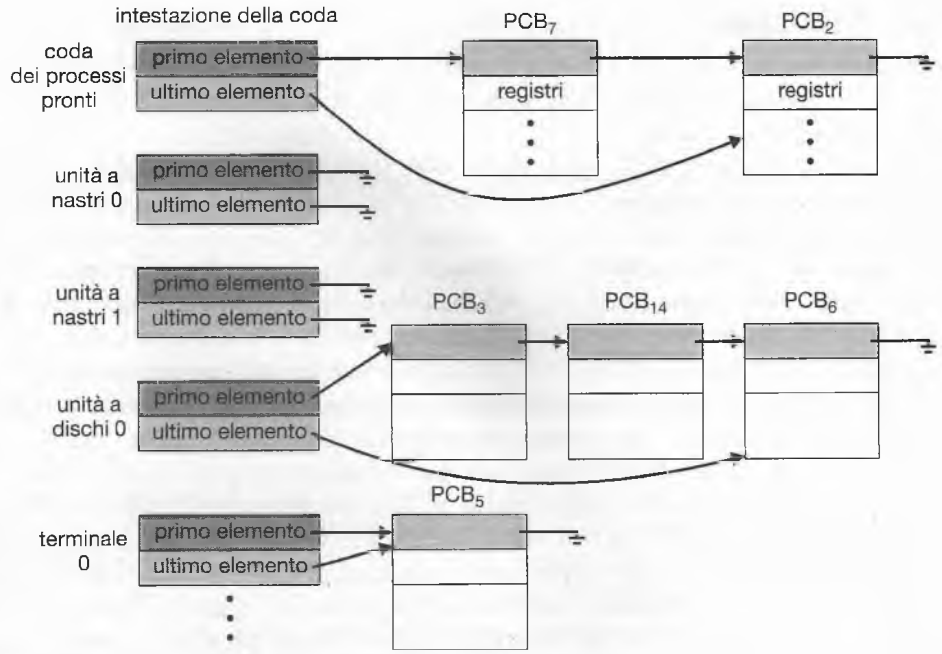

Figura 3.6 Coda dei processi pronti e diverse code di dispositivi I/O.

- il processo può emettere una richiesta di I/O e quindi essere inserito in una coda di I/O;
- il processo può creare un nuovo processo e attenderne la terminazione;
- il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio, il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella coda dei processi pronti. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene allontanato da tutte le code, rimosso il suo PCB e revocate le varie risorse.

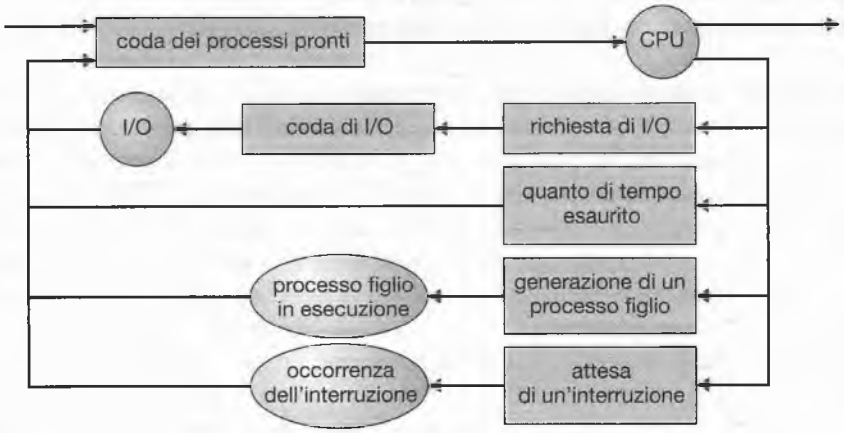

Figura 3.7 Diagramma di accodamento per lo scheduling dei processi.

3.2.2 Scheduler

Nel corso della sua esistenza, un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler**.

Spesso, in un sistema a lotti, accade che si sottopongano più processi di quanti se ne possano eseguire immediatamente. Questi lavori si trasferiscono in dispositivi di memoria secondaria, generalmente dischi, dove si tengono fino al momento dell'esecuzione (*spooling*). Lo **scheduler a lungo termine** (*job scheduler*), sceglie i lavori da questo insieme e li carica in memoria affinché siano eseguiti. Lo **scheduler a breve termine**, o **scheduler di CPU**, fa la selezione tra i lavori pronti per l'esecuzione e assegna la CPU a uno di loro.

Questi due scheduler si differenziano principalmente per la frequenza con la quale sono eseguiti. Lo scheduler a breve termine seleziona frequentemente un nuovo processo per la CPU. Il processo può essere in esecuzione solo per pochi millisecondi prima di passare ad attendere una richiesta di I/O. Poiché spesso si esegue almeno una volta ogni 100 millisecondi, lo scheduler a breve termine deve essere molto rapido: se impiegasse 10 millisecondi per decidere quale processo eseguire nei 100 millisecondi successivi, si userebbe, o per meglio dire si sprecherebbe, il 10/(100 + 10) = 9 per cento del tempo di CPU per il solo scheduling.

Lo scheduler a lungo termine, invece, si esegue con una frequenza molto inferiore; diversi minuti possono trascorrere tra la creazione di un nuovo processo e il successivo. Lo scheduler a lungo termine controlla il **grado di multiprogrammazione**, cioè il numero di processi presenti in memoria. Se è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità media con cui i processi abbandonano il sistema; quindi lo scheduler a lungo termine si può richiamare solo quando un processo abbandona il sistema. A causa del maggior intervallo che intercorre tra le esecuzioni, lo scheduler a lungo termine dispone di più tempo per scegliere un processo per l'esecuzione.

È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di I/O, o come avente una prevalenza d'elaborazione. Un **processo con prevalenza di I/O** (*I/O bound*) impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O. Un **processo con prevalenza d'elaborazione** (*CPU bound*), viceversa, richiede poche operazioni di I/O e impiega la maggior parte del proprio tempo nelle elaborazioni. È fondamentale che lo scheduler a lungo termine selezioni una buona **combinazione di processi** con prevalenza di I/O e con prevalenza d'elaborazione. Se tutti i processi fossero con prevalenza di I/O, la coda dei processi pronti sarebbe quasi sempre vuota e lo scheduler a breve termine resterebbe pressoché inattivo. Se tutti i processi fossero con prevalenza d'elaborazione, la coda d'attesa per l'I/O sarebbe quasi sempre vuota, i dispositivi non sarebbero utilizzati, e il sistema verrebbe nuovamente sbilanciato. Le prestazioni migliori sono date da una combinazione equilibrata di processi con prevalenza di I/O e processi con prevalenza d'elaborazione.

Esistono sistemi in cui lo scheduler a lungo termine può essere assente o minimo. Per esempio, i sistemi a partizione del tempo, come UNIX e Microsoft Windows, sono spesso privi di scheduler a lungo termine, e si limitano a caricare in memoria tutti i nuovi processi, gestiti dallo scheduler a breve termine. La stabilità di questi sistemi dipende dai limiti fisici degli stessi, come un numero limitato di terminali disponibili, oppure dall'autoregolamentazione insita nella natura degli utenti umani: quando ci si accorge che il rendimento della macchina scende a livelli inaccettabili si possono semplicemente chiudere alcune attività o la sessione di lavoro.

In alcuni sistemi operativi come quelli a partizione del tempo, si può introdurre un livello di scheduling intermedio. Questo **scheduler a medio termine** è rappresentato schema-

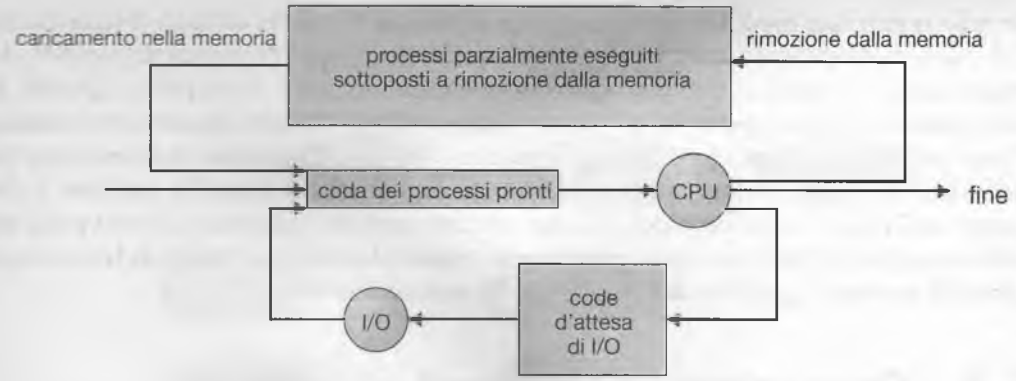

Figura 3.8 Aggiunta di scheduling a medio termine al diagramma di accodamento.

ticamente nella Figura 3.8. L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria (e dalla contesa attiva per la CPU), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta.

Questo schema si chiama **avvicendamento dei processi in memoria** – o, più in breve, **avvicendamento** (*swapping*). Il processo viene rimosso e successivamente caricato in memoria dallo scheduler a medio termine. L'avvicendamento dei processi in memoria può servire a migliorare la combinazione di processi, oppure a liberare una parte della memoria se un cambiamento dei requisiti di memoria ha impegnato eccessivamente la memoria disponibile. L'avvicendamento dei processi in memoria è illustrato nel Capitolo 8.

3.2.3 Cambio di contesto

Come spiegato nel Paragrafo 1.2.1, sono le interruzioni a indurre il sistema a sospendere il lavoro attuale della CPU per eseguire routine del kernel. Le interruzioni sono eventi comuni nei sistemi a carattere generale. In presenza di una interruzione, il sistema deve salvare il **contesto** del processo corrente, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione. Il contesto è rappresentato all'interno del PCB del processo, e comprende i valori dei registri della CPU, lo stato del processo (si veda la Figura 3.2), e informazioni relative alla gestione della memoria. In termini generali, si esegue un **salvataggio dello stato** corrente della CPU, sia che essa esegua in modalità utente o in modalità di sistema; in seguito, si attuerà un corrispondente **ripristino dello stato** per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

Il passaggio della CPU a un nuovo processo implica la registrazione dello stato del processo vecchio e il caricamento dello stato precedentemente registrato del nuovo processo. Questa procedura è nota col nome di **cambio di contesto** (*context switch*). Nell'evenienza di un cambio di contesto, il sistema salva nel suo PCB il contesto del processo uscente, e carica il contesto del processo subentrante, salvato in precedenza. Il cambio di contesto comporta un calo delle prestazioni, perché il sistema esegue solo operazioni volte alla corretta gestione dei processi, e non alla computazione. Il tempo necessario varia da sistema a sistema, dipendendo dalla velocità della memoria, dal numero di registri da copiare, e dall'esistenza di istruzioni macchina appropriate (per esempio, una singola istruzione per caricare o trasferire in memoria tutti i registri). In genere si tratta di qualche millisecondo.

La durata del cambio di contesto dipende molto dall'architettura; per esempio, alcune CPU (come la Sun UltraSPARC) offrono più gruppi di registri, quindi il cambio di contesto

prevede la semplice modifica di un puntatore al gruppo di registri corrente. Naturalmente, se il numero dei processi attivi è maggiore di quello dei gruppi di registri disponibili, il sistema rimedia copiando i dati dei registri nella e dalla memoria, come prima. Quindi, più complesso è il sistema operativo, più lavoro si deve svolgere durante un cambio di contesto. Come vedremo nel Capitolo 8, l'uso di tecniche complesse di gestione della memoria può richiedere lo spostamento di ulteriori dati a ogni cambio di contesto. Per esempio, si deve preservare lo spazio d'indirizzi del processo corrente mentre si prepara lo spazio per il processo successivo. Il modo in cui si preserva tale spazio e la relativa quantità di lavoro dipendono dal metodo di gestione della memoria del sistema operativo.

3.3 Operazioni sui processi

Nella maggior parte dei sistemi i processi si possono eseguire in modo concorrente, e si devono creare e cancellare dinamicamente; a tal fine il sistema operativo deve offrire un meccanismo che permetta di creare e terminare un processo. Nel presente paragrafo si esplorano quei meccanismi coinvolti nella creazione dei processi, in particolare rispetto ai sistemi UNIX e Windows.

3.3.1 Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita chiamata di sistema (`create_process`). Il processo creante si chiama processo **genitore**, mentre il nuovo processo si chiama processo **figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero** di processi.

La maggior parte dei sistemi operativi, compresi UNIX e la famiglia Windows, identifica un processo per mezzo di un numero univoco, detto **identificatore del processo** o **pid** (*process identifier*). Si tratta solitamente di un intero. La Figura 3.9 mostra un albero dei processi del sistema operativo Solaris, con il nome e il pid di ogni processo. In questo sistema, il processo alla radice dell'albero è `sched`, il cui pid è 0. `sched` ha svariati figli, fra cui `pageout` e `fsflush`. Questi processi sono responsabili della gestione della memoria e del file system. Il processo `sched` genera anche il processo `init`, che funge da genitore di tutti i processi utenti. La figura mostra due figli di `init`, `inetd` e `dtlogin`. Il primo gestisce i servizi di rete come `telnet` e `ftp`; il secondo rappresenta lo schermata di accesso al sistema. Quando un utente acceda al sistema, `dtlogin` crea una sessione X-windows (`Xsession`), che a sua volta genera il processo `sdt_shel`, al di sotto del quale risiede la shell dell'utente – per esempio, `csh`, ossia C-shell. È da tale interfaccia che l'utente invoca vari processi figli, come per esempio i comandi `ls` e `cat`. Dalla figura si può anche notare un processo `csh` con pid pari a 7778, che rappresenta un utente che abbia avuto accesso al sistema tramite `telnet`. Questo utente ha avviato il browser Netscape (pid 7785) e l'editor emacs (pid 8105).

Nei sistemi UNIX si può ottenere l'elenco dei processi tramite il comando `ps`. Digitando `ps -el` si otterranno informazioni complete su tutti i processi attualmente attivi nel sistema; è facile costruire un albero come quello nella figura identificando ricorsivamente i processi genitore fino a giungere a `init`.

In generale, per eseguire il proprio compito, un processo necessita di alcune risorse (tempo d'elaborazione, memoria, file, dispositivi di I/O). Quando un processo crea un sottoprocesso, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo

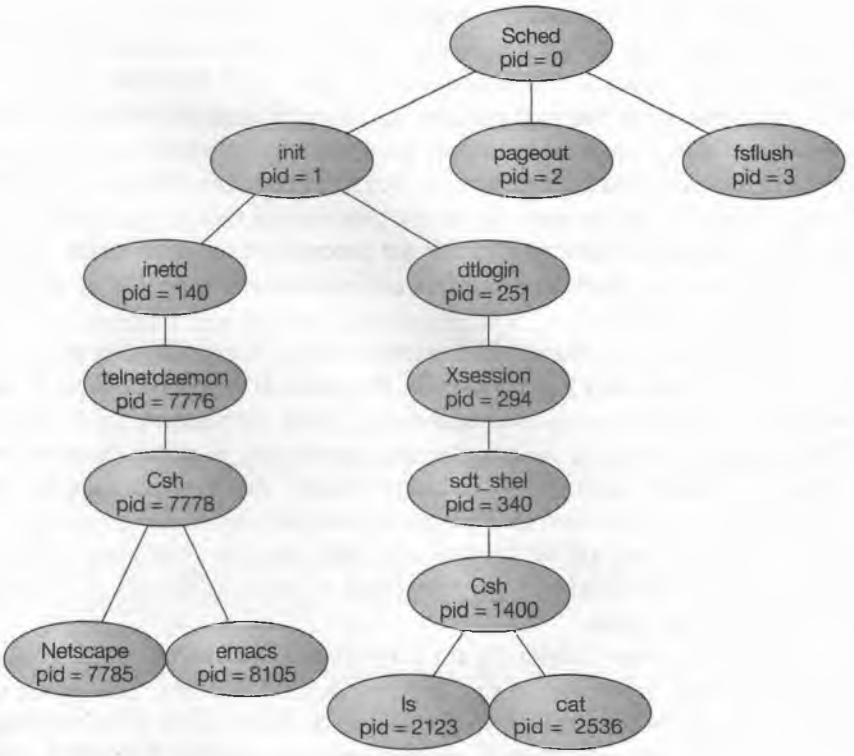

Figura 3.9 Esempio di albero dei processi di Solaris.

genitore. Il processo genitore può avere la necessità di spartire le proprie risorse tra i suoi processi figli, oppure può essere in grado di condividerne alcune, come la memoria o i file, tra più processi figli. Limitando le risorse di un processo figlio a un sottoinsieme di risorse del processo genitore, si può evitare che un processo sovraccarichi il sistema creando troppi sottoprocessi.

Quando si crea un processo, esso ottiene, oltre alle varie risorse fisiche e logiche, i dati di inizializzazione che il processo genitore può passare al processo figlio. Per esempio, si consideri un processo che serva a mostrare i contenuti di un file – diciamo, il file *img.jpg* – sullo schermo di un terminale. Esso riceverà dal genitore, al momento della sua creazione, il nome del file *img.jpg* in ingresso, che userà per aprire il file; potrà anche ricevere il nome del dispositivo al quale inviare i dati in uscita. Alcuni sistemi operativi passano anche risorse ai processi figli, nel qual caso il nostro processo potrebbe ottenere come risorse due nuovi file aperti, cioè *img.jpg* e il dispositivo terminale, potendo così semplicemente trasferire il dato fra i due.

Quando un processo ne crea uno nuovo, per quel che riguarda l'esecuzione ci sono due possibilità:

1. il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli;
2. il processo genitore attende che alcuni o tutti i suoi processi figli terminino.

Ci sono due possibilità anche per quel che riguarda lo spazio d'indirizzi del nuovo processo:

1. il processo figlio è un duplicato del processo genitore;
2. nel processo figlio si carica un nuovo programma.

Nel sistema operativo UNIX, per esempio, ogni processo è identificato dal proprio identificatore di processo, un intero unico. Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare senza difficoltà con il proprio processo figlio. Entrambi i processi (genitore e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema `fork()`, con una differenza: la chiamata di sistema `fork()` riporta il valore zero nel nuovo processo (il figlio), ma riporta l'identificatore del processo figlio (il PID diverso da zero) nel processo genitore. Tramite il valore riportato del PID, i due processi possono procedere nell'esecuzione "sapendo" qual è il processo padre e qual è il processo figlio.

Generalmente, dopo una chiamata di sistema `fork()`, uno dei due processi impiega una chiamata di sistema `exec()` per sostituire lo spazio di memoria del processo con un nuovo programma. La chiamata di sistema `exec()` carica in memoria un file binario, cancellando l'immagine di memoria del programma contenente la stessa chiamata di sistema `exec()`, quindi avvia la sua esecuzione. In questo modo i due processi possono comunicare e quindi procedere in modo diverso. Il processo genitore può anche generare più processi figli, oppure, se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la chiamata di sistema `wait()` per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio.

Il programma scritto in C della Figura 3.10 illustra le chiamate di sistema UNIX descritte precedentemente. Si ottengono due processi distinti ciascuno dei quali è un'istanza d'esecuzione dello stesso programma. Il valore assegnato alla variabile `pid` è zero nel processo figlio, e un numero intero maggiore di zero nel processo genitore. Usando la chiamata di sistema `execlp()` – una versione della chiamata di sistema `exec()` – il processo figlio sovrappone il proprio spazio d'indirizzi con il comando `/bin/ls` di UNIX (che si usa per ottenere l'elenco del contenuto di una directory). Impiegando la chiamata di sistema `wait()`, il processo genitore attende che il processo figlio termini. Quando ciò accade, il processo genitore chiude la propria fase d'attesa dovuta alla chiamata di sistema `wait()` e termina usando la chiamata di sistema `exit()`. Questo passaggio è illustrato nella Figura 3.11.

Come altro esempio, consideriamo la creazione dei processi in Windows. Nella API Win32 la funzione `CreateProcess()` è simile alla `fork()` di UNIX, poiché serve a generare un figlio da un processo genitore. Mentre però `fork()` passa in eredità al figlio lo spazio degli indirizzi del genitore, `CreateProcess()` richiede il caricamento di un programma specificato nello spazio degli indirizzi del processo figlio al momento della sua creazione. Inoltre, mentre `fork()` non richiede parametri, `CreateProcess()` se ne aspetta non meno di dieci. Il programma nella Figura 3.12, scritto in C, illustra la funzione `CreateProcess()`; essa genera un processo figlio che carica l'applicazione `mspaint.exe`. In questo esempio, i dieci parametri passati a `CreateProcess()` hanno in molti casi il loro valore di default.

I lettori interessati alla creazione e gestione dei processi nella API Win32 possono consultare i riferimenti bibliografici alla fine del capitolo.

Due dei parametri passati a `CreateProcess()` sono istanze delle strutture `STARTUPINFO` e `PROCESS_INFORMATION`. La prima specifica molte proprietà del nuovo processo, come la dimensione della finestra e il suo aspetto, e i riferimenti – detti anche **handle** ("maniglie") – ai file di input e output standard. La seconda contiene un handle e gli identificatori per il nuovo processo e il suo thread. La funzione `ZeroMemory()` è invocata per allocare memoria per le due strutture prima di passare a `CreateProcess()`.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* genera un nuovo processo */
    pid = fork();

    if (pid < 0) { /* errore */
      fprintf(stderr, "generazione del nuovo processo fallita");
      return 1;
    }
    else if (pid == 0) { /* processo figlio */
      execlp("/bin/ls", "ls", NULL);
    }
    else {/* processo genitore */
      /* il genitore attende il completamento del figlio */
      wait(NULL);
      printf("il processo figlio ha terminato");
      }

      return 0;
}
```

Figura 3.10 Creazione di un processo separato utilizzando la chiamata di sistema `fork()`.

I primi due parametri passati a `CreateProcess()` sono il nome dell'applicazione e i parametri della riga di comando. Se il nome dell'applicazione è `WaitForSingleObject()` (come nell'esempio della figura), è il parametro della riga di comando a specificare quale applicazione caricare. Nell'esempio, si carica l'applicazione *mspaint.exe* di Microsoft Windows. Al di là dei primi due parametri, l'esempio usa i parametri di default per far ereditare gli handle del processo e del thread, e per specificare l'assenza di flag di creazione. Il figlio adotta inoltre il blocco ambiente del genitore e la sua directory iniziale. Infine, si passano i due

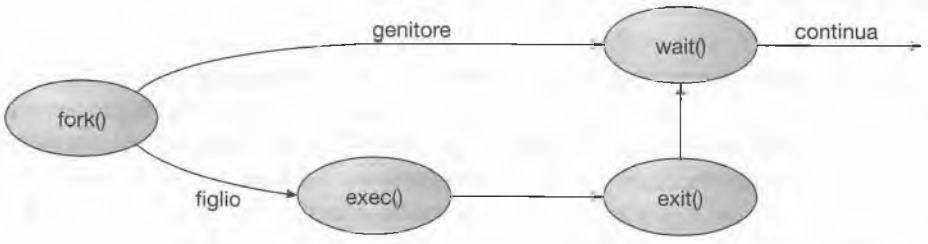

Figura 3.11 Generazione di un processo per mezzo della chiamata di sistema `fork()`.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // alloca la memoria
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // genera processo figlio
    if (!CreateProcess(NULL, // usa riga di comando
     "C:\\WINDOWS\\system32\\mspaint.exe", // riga di comando
     NULL, // non eredita l'handle del processo
     NULL, // non eredita l'handle del thread
     FALSE, // disattiva l'ereditarieta' degli handle
     0, // nessun flag di creazione
     NULL, // usa il blocco ambiente del genitore
     NULL, // usa la directory esistente del genitore
     &si,
     &pi))
    {
      fprintf(stderr, "generazione del nuovo processo fallita");
      return -1
    }
    // il genitore attende il completamento del figlio
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("il processo figlio ha terminato");

    // rilascia gli handle
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figura 3.12 Creazione di un nuovo processo con la API Win32.

puntatori alle strutture `STARTUPINFO` e `PROCESS_INFORMATION` create all'inizio. Nella precedente Figura 3.10, il processo genitore attende la terminazione del figlio invocando la chiamata di sistema `wait()`. L'equivalente per Win32 è `WaitForSingleObject()`, a cui si passa l'handle del processo figlio – `pi.hProcess` – del cui completamento si è in attesa. A terminazione del figlio avvenuta, il controllo ritorna al processo genitore, al punto immediatamente successivo alla chiamata `WaitForSingleObject()`.

3.3.2 Terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema `exit()`; a questo punto, il processo figlio può riportare alcuni dati al processo genitore, che li riceve attraverso la chiamata di sistema `wait()`. Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'I/O, sono liberate dal sistema operativo.

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un'opportuna chiamata di sistema (per esempio `TerminateProcess()` in Win32). Generalmente solo il genitore del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un genitore deve conoscere le identità dei propri figli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi, tra i quali i seguenti.

- Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate. Ciò richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli.
- Il compito assegnato al processo figlio non è più richiesto.
- Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza.

In alcuni sistemi, fra i quali VMS, se un processo termina si devono terminare anche i suoi figli, indipendentemente dal fatto che la terminazione del genitore sia stata normale o anormale. Si parla di **terminazione a cascata**, una procedura avviata di solito dal sistema operativo.

Per illustrare un esempio di esecuzione e terminazione di un processo, si consideri che nel sistema operativo UNIX un processo può terminare per mezzo della chiamata di sistema `exit()`, e il suo processo genitore può attendere l'evento per mezzo della chiamata di sistema `wait()`. Quest'ultima riporta l'identificatore di un processo figlio che ha terminato l'esecuzione, sicché il genitore può stabilire quale tra i suoi processi figli l'abbia terminata. Se un processo genitore termina, tutti i suoi processi figli sono affidati al processo `init()`, che assume il ruolo di nuovo genitore, cui i processi figli possono riportare i risultati delle proprie attività.

3.4 Comunicazione tra processi

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è **indipendente** se non può influire su altri processi del sistema o subirne l'influsso. Chiaramente, un processo che non condivide dati (temporanei o permanenti) con altri processi è indipendente. D'altra parte un processo è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivide dati con altri processi è un processo cooperante.

Un ambiente che consente la cooperazione tra processi può essere utile per diverse ragioni.

- **Condivisione d'informazioni.** Poiché più utenti possono essere interessati alle stesse informazioni (per esempio un file condiviso), è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse.
- **Accelerazione del calcolo.** Alcune attività d'elaborazione sono realizzabili più rapidamente se si suddividono in sottoattività eseguibili in parallelo. Un'accelerazione di questo tipo è ottenibile solo se il calcolatore dispone di più elementi capaci di attività d'elaborazione (come più CPU o canali di I/O).
- **Modularità.** Può essere necessaria la costruzione di un sistema modulare, che suddivida le funzioni di sistema in processi o thread distinti (si veda il Capitolo 2).
- **Convenienza.** Anche un solo utente può avere la necessità di compiere più attività contemporaneamente; per esempio, può eseguire in parallelo le operazioni di scrittura, stampa e compilazione.

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi** (IPC, *interprocess communication*). I modelli fondamentali della comunicazione tra processi sono due: (1) a **memoria condivisa** e (2) a **scambio di messaggi**. Nel modello a memoria condivisa, si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona. Nel secondo tipo di modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti. I due modelli sono messi a confronto nella Figura 3.13.

Nei sistemi operativi sono diffusi entrambi i modelli; a volte coesistono in un unico sistema. Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti. La memoria condivisa massimizza l'efficienza della comunicazione, ed è più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono gestiti alla stregua di ordinari accessi in memoria che non richiedono l'assistenza del kernel. Nel seguito di questo paragrafo esploriamo i due modelli IPC in maggiore dettaglio.

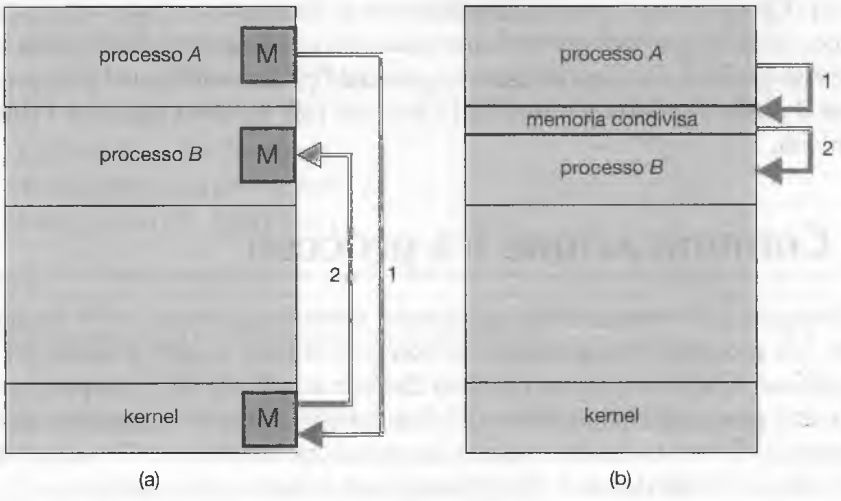

Figura 3.13 Modelli di comunicazione tra processi basati su (a) scambio di messaggi, e (b) condivisione della memoria.

3.4.1 Sistemi a memoria condivisa

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca: gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi. Si ricordi che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di altri processi. La condivisione della memoria richiede che due o più processi raggiungano un accordo per superare questo limite, in modo da poter comunicare tramite scritture e letture dell'area condivisa. Il tipo e la collocazione dei dati sono determinati dai processi, e rimangono al di fuori del controllo del sistema operativo. I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente.

Per illustrare il concetto di cooperazione tra processi, si consideri il problema del produttore/consumatore; tale problema è un usuale paradigma per processi cooperanti. Un processo **produttore** produce informazioni che sono consumate da un processo **consumatore**. Un compilatore, per esempio, può produrre del codice assembly consumato da un assemblatore; quest'ultimo, a sua volta, può produrre moduli oggetto consumati dal caricatore. Il problema del produttore/consumatore è anche un'utile metafora del paradigma client-server. Si pensa in genere al server come al produttore e al client come al consumatore. Per esempio, un server web produce (ossia, fornisce) pagine HTML e immagini, consumate (ossia, lette) dal client, cioè il browser web che le richiede.

Una possibile soluzione del problema del produttore/consumatore si basa sulla memoria condivisa. L'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi. Il produttore potrà allora produrre un'unità, e il consumatore consumarne un'altra. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità non ancora prodotta.

Sono utilizzabili due tipi di buffer. Quello **illimitato** non pone limiti alla dimensione del buffer. Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne. Il problema del produttore e del consumatore con **buffer limitato** presuppone l'esistenza di una dimensione fissa del buffer in questione. In questo caso, il consumatore deve attendere che il buffer sia vuoto; viceversa, il produttore deve attendere che il buffer sia pieno.

Consideriamo più attentamente come il buffer limitato sia utilizzabile per la condivisione di memoria tra processi. Le variabili seguenti risiedono in una zona di memoria condivisa sia dal produttore sia dal consumatore.

```
#define DIM_BUFFER 10

typedef struct {
   . . .
}elemento;

elemento buffer[DIM_BUFFER];
int inserisci = 0;
int preleva = 0;
```

Il buffer condiviso è realizzato come un array circolare con due puntatori logici: `inserisci` e `preleva`. La variabile `inserisci` indica la successiva posizione libera nel buffer; `preleva` indica la prima posizione piena nel buffer. Il buffer è vuoto se `inserisci == preleva`; è pieno se `((inserisci + 1) % DIM_BUFFER) == preleva`.

```
elemento appena_Prodotto;

while (true) {
    /* produce un elemento in appena_Prodotto */
    while (((inserisci + 1) % DIM_BUFFER) == preleva)
      ; /* non fa niente */
    buffer[inserisci] = appena_Prodotto;
    inserisci = (inserisci + 1) % DIM_BUFFER;
}
```

Figura 3.14 Processo produttore.

```
elemento da_Consumare;

while (true) {
   while (inserisci == preleva)
      ; /* non fa niente */

   da_Consumare = buffer[preleva];
   preleva = (preleva + 1) % DIM_BUFFER;
   /* consuma l'elemento in da_Consumare */
}
```

Figura 3.15 Processo consumatore.

Di seguito si riporta il codice per i processi produttore e consumatore (illustrato, rispettivamente, nella Figura 3.14 e 3.15). Il processo produttore ha una variabile locale `appena_Prodotto` contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile locale `da_Consumare` in cui si memorizza l'elemento da consumare.

Questo metodo ammette un massimo di `DIM_BUFFER-1` oggetti contemporaneamente presenti nel buffer. Proponiamo come esercizio per il lettore la stesura di un algoritmo che permetta la presenza contemporanea di `DIM_BUFFER` oggetti. Nel Paragrafo 3.5.1 si illustrerà la API POSIX per la memoria condivisa.

Una questione ignorata dalla precedente analisi è il caso in cui sia il produttore sia il consumatore tentano di accedere al buffer concorrentemente. Nel Capitolo 6 si vedrà come sia possibile implementare efficacemente la sincronizzazione tra processi cooperanti nel modello a memoria condivisa.

3.4.2 Sistemi a scambio di messaggi

Nel Paragrafo 3.4.1 si è parlato di come può avvenire la comunicazione tra processi cooperanti in un ambiente a memoria condivisa. Per essere applicato, lo schema proposto richiede che tali processi condividano l'accesso a una zona di memoria e che il codice per la realizzazione e la gestione della memoria condivisa sia scritto esplicitamente dal programmatore che crea l'applicazione. Un altro modo in cui il sistema operativo può ottenere i medesimi risultati consiste nel fornire ai processi appositi strumenti per lo scambio di messaggi.

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio degli indirizzi. È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete. Per esempio, una **chat** sul Web potrebbe essere implementata tramite scambio di messaggi fra i vari partecipanti.

Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni: `send` (cioè, "invia messaggio") e `receive` (cioè, "ricevi messaggio"). I messaggi possono avere lunghezza fissa o variabile. Nel primo caso, l'implementazione a livello del sistema è elementare, ma programmare applicazioni diviene più complicato. Nel secondo caso, l'implementazione del meccanismo è più complessa, mentre la programmazione utente risulta semplificata. Compromessi di questo tipo si riscontrano spesso nella progettazione dei sistemi operativi.

Se i processi *P* e *Q* vogliono comunicare, devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un **canale di comunicazione** (*communication link*), realizzabile in molti modi. In questo paragrafo non si tratta della realizzazione fisica del canale (come la memoria condivisa, i bus o le reti, illustrati nel Capitolo 16) ma della sua realizzazione logica. Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni `send()` e `receive()`:

- comunicazione diretta o indiretta;
- comunicazione sincrona o asincrona;
- gestione automatica o esplicita del buffer.

Le questioni legate a ciascuna di tali caratteristiche vengono illustrate in seguito.

3.4.2.1 Nominazione

Per comunicare, i processi devono disporre della possibilità di far riferimento ad altri processi; a tale scopo è possibile servirsi di una comunicazione diretta oppure indiretta.

Con la **comunicazione diretta**, ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione. In questo schema le funzioni primitive `send()` e `receive()` si definiscono come segue:

- send(P, messaggio), invia messaggio al processo P;
- receive(Q, messaggio), riceve, in messaggio, un messaggio dal processo Q.

All'interno di questo schema, un canale di comunicazione ha le seguenti caratteristiche:

- tra ogni coppia di processi che intendono comunicare si stabilisce automaticamente un canale; i processi devono conoscere solo la reciproca identità;
- un canale è associato esattamente a due processi.

Esiste esattamente un canale tra ciascuna coppia di processi.

Questo schema ha una *simmetria* nell'indirizzamento, vale a dire che per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda. Una variante di questo schema si avvale dell'*asimmetria* dell'indirizzamento: soltanto il trasmittente nomina il ricevente, mentre il ricevente non deve nominare il trasmittente. In questo schema le primitive `send()` e `receive()` si definiscono come segue:

- send(P, messaggio), invia messaggio al processo P;
- receive(id, messaggio), riceve, in messaggio, un messaggio da qualsiasi processo; nella variabile id si riporta il nome del processo con cui è avvenuta la comunicazione.

Entrambi gli schemi, simmetrico e asimmetrico, hanno lo svantaggio di una limitata modularità delle risultanti definizioni dei processi. La modificazione del nome di un processo può infatti implicare la necessità di un riesame di tutte le altre definizioni dei processi, individuando tutti i riferimenti al vecchio nome allo scopo di sostituirli con il nuovo. In generale, tali **cablature** (*hard coding*) di informazioni nel codice sono meno vantaggiose di soluzioni indirette o parametriche, approfondite di seguito.

Con la **comunicazione indiretta**, i messaggi s'inviano a delle **porte** (dette anche *mailbox*), che li ricevono. Una porta si può considerare in modo astratto come un oggetto in cui i processi possono introdurre e prelevare messaggi, ed è identificata in modo unico. Per l'identificazione di una porta le code di messaggi POSIX usano per esempio un valore intero. In questo schema un processo può comunicare con altri processi tramite un certo numero di porte. Due processi possono comunicare solo se condividono una porta. Le primitive `send()` e `receive()` si definiscono come segue:

- send(A, messaggio), invia messaggio alla porta A;
- receive(A, messaggio), riceve, in messaggio, un messaggio dalla porta A.

In questo schema un canale di comunicazione ha le seguenti caratteristiche:

- tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa porta;
- un canale può essere associato a più di due processi;
- tra ogni coppia di processi comunicanti possono esserci più canali diversi, ciascuno corrispondente a una porta.

A questo punto, si supponga che i processi P_1, P_2 e P_3 condividano la porta A. Il processo P_1 invia un messaggio ad A, mentre sia P_2 sia P_3 eseguono una `receive()` da A. Sorge il problema di sapere quale processo riceverà il messaggio. La soluzione dipende dallo schema prescelto:

- si può fare in modo che un canale sia associato al massimo a due processi;
- si può consentire l'esecuzione di un'operazione `receive()` a un solo processo alla volta;
- si può consentire al sistema di decidere arbitrariamente quale processo riceverà il messaggio (il messaggio sarà ricevuto da P_2 o da P_3, ma non da entrambi). Il sistema può anche definire un algoritmo per selezionare quale processo riceverà il messaggio (specificando cioè uno schema, detto *round robin*, secondo il quale i processi ricevono i messaggi a turno) e può comunicare l'identità del ricevente al trasmittente.

Una porta può appartenere al processo o al sistema. Se appartiene a un processo, cioè fa parte del suo spazio d'indirizzi, occorre distinguere ulteriormente tra il proprietario, che può soltanto ricevere messaggi tramite la porta, e l'utente, che può solo inviare messaggi alla porta. Poiché ogni porta ha un unico proprietario, non può sorgere confusione su chi debba ricevere un messaggio inviato a una determinata porta. Quando un processo che possiede una porta termina, questa scompare, e qualsiasi processo che invii un messaggio alla porta di un processo già terminato ne deve essere informato.

D'altra parte, una porta posseduta dal sistema operativo è indipendente e non è legata ad alcun processo particolare. Il sistema operativo offre un meccanismo che permette a un processo le seguenti operazioni:

- creare una nuova porta;
- inviare e ricevere messaggi tramite la porta;
- rimuovere una porta.

Il processo che crea una nuova porta è il proprietario predefinito della porta, inizialmente è l'unico processo che può ricevere messaggi attraverso questa porta. Tuttavia, il diritto di proprietà e il diritto di ricezione si possono passare ad altri processi per mezzo di idonee chiamate di sistema. Naturalmente questa disposizione potrebbe dar luogo alla creazione di più riceventi per ciascuna porta.

3.4.2.2 Sincronizzazione

La comunicazione tra processi avviene attraverso chiamate delle primitive `send()` e `receive()`. Ci sono diverse possibilità nella definizione di ciascuna primitiva. Lo scambio di messaggi può essere **sincrono** (o **bloccante**) oppure **asincrono** (o **non bloccante**).

- **Invio sincrono.** Il processo che invia il messaggio si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio.
- **Invio asincrono.** Il processo invia il messaggio e riprende la propria esecuzione.
- **Ricezione sincrona.** Il ricevente si blocca nell'attesa dell'arrivo di un messaggio.
- **Ricezione asincrona.** Il ricevente riceve un messaggio valido oppure un valore nullo.

È possibile anche avere diverse combinazioni di `send()` e `receive()` tra quelle illustrate sopra. Se le primitive `send()` e `receive()` sono entrambe bloccanti si parla di **rendezvous** tra mittente e ricevente. È banale dare soluzione al problema del produttore e del consumatore tramite le primitive bloccanti `send()` e `receive()`. Il produttore, infatti, si limita a invocare `send()` e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore richiama `receive()`, bloccandosi fino all'arrivo di un messaggio.

Come si avrà modo di notare durante la lettura del testo, i concetti di sincronia e asincronia compaiono spesso negli algoritmi per l'I/O dei sistemi operativi.

3.4.2.3 Code di messaggi

Se la comunicazione è diretta o indiretta, i messaggi scambiati tra processi comunicanti risiedono in code temporanee. Fondamentalmente esistono tre modi per realizzare queste code.

- **Capacità zero.** La coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio.
- **Capacità limitata.** La coda ha lunghezza finita n, quindi al suo interno possono risiedere al massimo n messaggi. Se la coda non è piena, quando s'invia un nuovo messaggio, quest'ultimo è posto in fondo alla coda; il messaggio viene copiato oppure si tiene un puntatore a quel messaggio. Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale ha tuttavia una capacità limitata; se è pieno, il trasmittente deve fermarsi nell'attesa che ci sia spazio disponibile nella coda.
- **Capacità illimitata.** La coda ha una lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

Il caso con capacità zero è talvolta chiamato sistema a scambio di messaggi senza memorizzazione transitoria (*no buffering*); gli altri due, sistemi con memorizzazione transitoria automatica (*automatic buffering*).

3.5 Esempi di sistemi per la IPC

In questo paragrafo analizzeremo tre sistemi per la comunicazione fra processi: la API POSIX basata sulla memoria condivisa, lo scambio di messaggi nel sistema operativo Mach, e la interessante mistura di queste due soluzioni adottata da Windows XP.

3.5.1 Un esempio: memoria condivisa secondo POSIX

Lo standard POSIX prevede svariati meccanismi per la IPC, compresa la condivisione della memoria e lo scambio di messaggi. Qui illustreremo la API POSIX per la condivisione della memoria.

Per prima cosa, il processo deve allocare un segmento condiviso di memoria tramite la chiamata di sistema `shmget()`; l'acronimo deriva da *shared memory get*, ossia "acquisisci memoria condivisa". Per esempio, nella riga

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

il primo parametro specifica la chiave o l'identificatore del segmento condiviso. Se esso vale `IPC_PRIVATE` il sistema alloca un nuovo segmento di memoria condivisa. Il secondo parametro è la dimensione in byte del segmento. Il terzo parametro stabilisce il modo d'accesso al segmento – in lettura, in scrittura, o in entrambi i modi. Il valore `S_IRUSR | S_IWUSR` indica che il proprietario del segmento può accedervi sia in lettura sia in scrittura. Se la chiamata a `shmget()` va a buon fine, il valore restituito è un intero che identifica il segmento di memoria allocato: i processi che vorranno condividere questo segmento, dovranno richiederlo per mezzo del suo identificatore.

I processi che vogliano accedere a un segmento di memoria condivisa devono annetterlo al loro spazio degli indirizzi tramite una chiamata a `shmat()`, acronimo di *shared memory attach* – "annetti memoria condivisa". Anch'essa richiede tre parametri, di cui il primo è l'identificatore intero del segmento da annettere, e il secondo è un puntatore che indica in che punto della memoria va annesso il segmento condiviso; se tale puntatore vale `NULL`, la scelta del punto sarà demandata al sistema operativo. Il terzo parametro è un selettore di modo d'accesso. Quando esso vale zero, il segmento è annesso con modalità sola lettura, mentre se il selettore è impostato (cioè, vale > 0), l'accesso è consentito sia in lettura sia in scrittura. Se l'invocazione

```
shared_memory = (char *) shmat(id, NULL, 0);
```

ha successo, `shmath()` restituisce un puntatore alla prima locazione di memoria ove è stato annesso il segmento condiviso.

Usando questo puntatore, il processo può dunque accedere alla memoria annessa nello stesso modo in cui accederebbe ordinariamente alla memoria. Nell'esempio, `shmat()` restituisce un puntatore a una stringa di caratteri; si può quindi scrivere nell'area condivisa in questo modo:

```
sprintf(shared_memory, "Scrittura in memoria condivisa");
```

Gli altri processi che condividono il segmento rileveranno la modifica in memoria.

Di solito, la procedura seguita dai processi per condividere memoria già allocata è quella di annettere un segmento preesistente, e poi accedervi (eventualmente modificandolo). Quando un dato processo non ha più bisogno di condividere un segmento di memoria, lo elimina dal suo spazio degli indirizzi tramite

```
shmdt(shared_memory);
```

dove `shared_memory` è il puntatore al segmento di memoria da eliminare. Infine, un segmento condiviso può essere eliminato dall'intero sistema tramite `shmctl()`, cui si passa l'identificatore del segmento insieme al selettore `IPC_RMID`.

Il programma illustrato nella Figura 3.16 mostra la API POSIX che abbiamo preso in esame. Esso alloca un segmento condiviso di 4096 byte, e scrive il messaggio `Ciao` al suo interno. Dopo aver stampato il contenuto della memoria condivisa, il programma lo elimina prima dal proprio spazio degli indirizzi e poi dall'intero sistema. Altri esercizi sulla API POSIX per la condivisione della memoria si trovano alla fine del capitolo.

3.5.2 Un esempio: Mach

Come esempio di sistema operativo basato sullo scambio di messaggi, consideriamo il sistema operativo Mach, sviluppato alla Carnegie Mellon University. Abbiamo già presentato questo sistema operativo nel Capitolo 2, come parte di Mac OS X. Il suo kernel consente la creazione e la soppressione di più *task*, simili ai processi, ma che hanno più thread di controllo. La maggior parte delle comunicazioni – compresa una gran parte delle chiamate di sistema e tutte le informazioni tra task – si compie per mezzo di *messaggi*. I messaggi s'inviano e si ricevono attraverso *porte*.

Anche le chiamate di sistema s'invocano per mezzo di messaggi. Al momento della creazione di ogni task si creano due porte speciali: la porta **Kernel** e la porta **Notify**. Il kernel usa la porta Kernel per comunicare con il task e notifica l'occorrenza di un evento alla porta Notify. Per il trasferimento dei messaggi sono necessarie solo tre chiamate di sistema: `msg_send()`, che invia un messaggio a una porta; `msg_receive()`, per ricevere un messaggio; `msg_rpc()`, per le **chiamate di procedure remote** (*remote procedure call*, RPC), che invia un messaggio e ne attende esattamente uno di risposta per il trasmittente (in questo modo la RPC riproduce l'usuale chiamata di procedura, ma può operare tra sistemi diversi, da cui il termine *remota*).

La chiamata di sistema `port_allocate()` crea una nuova porta e assegna lo spazio per la sua coda di messaggi. La dimensione massima predefinita di tale coda è di otto messaggi. Il task che crea la porta è il proprietario della stessa, e può accedervi per la ricezione dei messaggi. Solo un task alla volta può possedere una porta o ricevere da una porta ma, all'occorrenza, questi diritti si possono trasmettere anche ad altri task.

La porta ha inizialmente una coda di messaggi vuota e i messaggi si copiano nella porta nell'ordine in cui sono ricevuti: tutti i messaggi hanno la stessa priorità. Mach garantisce che più messaggi in arrivo dallo stesso trasmittente siano accodati nell'ordine d'arrivo (*first-in, first-out*, FIFO), ma non garantisce un ordinamento assoluto. Per esempio, i messaggi inviati da due trasmittenti possono essere accodati in un ordine qualsiasi.

Gli stessi messaggi sono composti da un'intestazione di lunghezza fissa, seguita da una porzione di dati di lunghezza variabile. L'intestazione contiene la lunghezza del messaggio e due nomi di porte, uno dei quali è il nome della porta cui s'invia il messaggio. Normalmente il thread trasmittente attende una risposta; il nome della porta del trasmittente è passato al task ricevente, che lo impiega come un "indirizzo del mittente".

La parte variabile del messaggio è composta da una lista di dati tipizzati. Ogni elemento della lista ha un tipo, una dimensione e un valore. Il tipo degli oggetti specificati nel messaggio è importante, poiché oggetti definiti dal sistema operativo, come diritti di proprietà o di ricezione, stati del task e segmenti di memoria, si possono inviare all'interno dei messaggi.

Anche le operazioni di trasmissione e ricezione sono piuttosto flessibili. Per esempio, quando s'invia un messaggio a una porta che non è già piena, lo si copia al suo interno e il

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* identificatore del segmento condiviso */
int segment_id;
/* puntatore al segmento condiviso */
char* shared_memory;
/* dimensione (in byte) del segmento condiviso */
const int size = 4096;

    /* alloca il segmento condiviso */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* annette il segmento condiviso */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* scrive un messaggio nel segmento condiviso */
    sprintf(shared_memory, "Ciao");

    /* stampa la stringa contenuta nel segmento condiviso */
    printf("*%s\n", shared_memory);

    /* elimina il segmento condiviso dal proprio spazio indirizzi */
    shmdt(shared_memory);

    /* elimina il segmento condiviso dal sistema */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Figura 3.16 Programma C che illustra la API POSIX per la condivisione della memoria.

thread trasmittente prosegue la sua esecuzione. Se la porta è piena, il thread trasmittente ha le seguenti possibilità.

1. Attendere indefinitamente che nella porta ci sia spazio.
2. Attendere al massimo *n* millisecondi.
3. Non attendere, ma rientrare immediatamente.
4. Memorizzare temporaneamente il messaggio. Un messaggio si può consegnare al sistema operativo anche se la porta che dovrebbe riceverlo è piena. Quando il messaggio può effettivamente essere messo nella porta, s'invia un avviso al trasmittente; per una porta piena può restare in sospeso un solo messaggio di questo tipo, in qualunque momento per un dato thread trasmittente.

L'ultima possibilità si usa per i task che svolgono servizi (*server task*), come i driver delle stampanti. Dopo aver portato a termine una richiesta, questi task possono aver bisogno d'inviare un'unica risposta al task che aveva richiesto il servizio, ma devono anche proseguire con altre richieste di servizi, anche se la porta di risposta per un client è piena.

Nell'operazione `receive()` occorre specificare da quale porta o insieme di porte debba provenire il messaggio. Un **insieme di porte** (*mailbox set*), dichiarato dal task, si tratta come se fosse un'unica porta. I thread di un task possono ricevere solo da un insieme di porte (o da una porta) per cui tale task ha il diritto di ricezione. Una chiamata di sistema `port_status()` restituisce il numero dei messaggi in una data porta. L'operazione di ricezione tenta di ricevere da (1) una qualsiasi tra le porte di un insieme; o (2) da una porta specifica (nominata). Se non è atteso alcun messaggio, il thread ricevente può attendere un massimo di *n* millisecondi o non attendere affatto.

Il sistema Mach è stato progettato per i sistemi distribuiti (descritti nei Capitoli dal 16 al 18), ma è adatto anche a sistemi con singola CPU. I problemi più gravi dei sistemi a scambio di messaggi sono generalmente dovuti alle scarse prestazioni dovute alla doppia operazione di copiatura dei messaggi dal trasmittente alla porta e quindi dalla porta al ricevente. Il sistema di messaggi di Mach cerca di evitare doppie operazioni di copiatura impiegando tecniche di gestione della memoria virtuale (Capitolo 9). Fondamentalmente Mach associa lo spazio d'indirizzi contenente il messaggio del trasmittente allo spazio d'indirizzi del ricevente. Il messaggio stesso non è mai effettivamente copiato, quindi le prestazioni del sistema migliorano notevolmente anche se solo nel caso di messaggi all'interno dello stesso sistema.

3.5.3 Un esempio: Windows XP

Il sistema operativo Windows XP è un esempio di moderno progetto che impiega la modularità per aumentare la funzionalità e diminuire il tempo necessario alla realizzazione di nuove caratteristiche. Windows XP gestisce più ambienti operativi o *sottosistemi*, con cui i programmi applicativi comunicano attraverso un meccanismo a scambio di messaggi; tali programmi si possono considerare *client* del *server* costituito dal sottosistema.

La funzione di scambio di messaggi di Windows XP è detta **chiamata di procedura locale** (*local procedure call*, LPC) e si usa per la comunicazione tra due processi presenti nello stesso calcolatore. È simile al meccanismo standard della chiamata di procedura remota, ma è ottimizzata per questo sistema. Come in Mach, nel sistema Windows XP s'impiega un oggetto porta per stabilire e mantenere una connessione tra due processi. Ogni client che invoca un sottosistema necessita di un canale di comunicazione che è fornito da un oggetto porta e che non è mai ereditato. Windows XP usa due tipi di porte: le porte di connessione e le porte di comunicazione. Sono essenzialmente identiche ma ricevono nomi diversi secondo il modo in cui si usano. Le porte di connessione sono *oggetti con nome* visibili a tutti i processi e forniscono alle applicazioni un modo per stabilire un canale di comunicazione (Capitolo 22). Tale comunicazione funziona come segue.

- Il client apre un handle per l'oggetto porta di connessione del sottosistema.
- Il client invia una richiesta di connessione.
- Il server crea due porte di comunicazione private e restituisce l'handle per una di loro al client.
- Il client e il server impiegano l'handle corrispondente di porta per inviare messaggi o *callback* ("chiamate di ritorno") e ascoltare le risposte.

Il sistema Windows XP si serve di tre tipi di tecniche di scambio di messaggi su una porta specificata dal client quando stabilisce il canale. La più semplice si usa per brevi messaggi (fi-

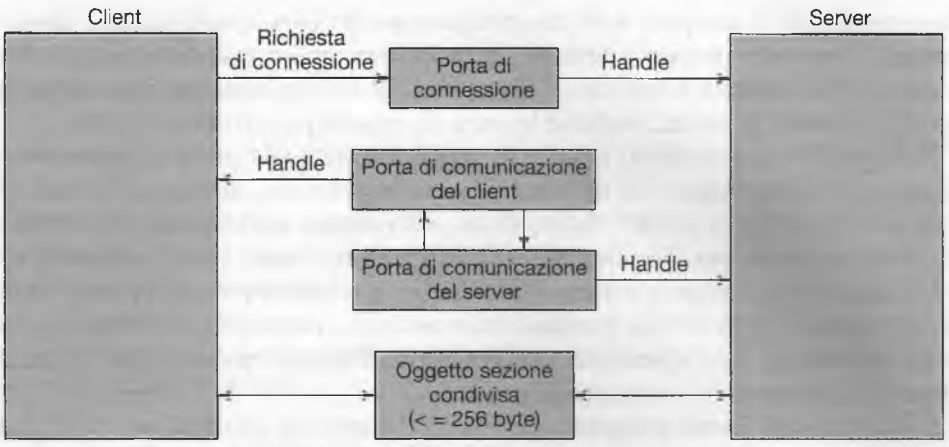

Figura 3.17 Chiamate di procedura locale in Windows XP.

no a 256 byte) e consiste nell'impiegare la coda dei messaggi della porta come una memoria intermedia per copiare il messaggio da un processo all'altro.

Se il client deve inviare un messaggio più lungo, il trasferimento avviene tramite un **oggetto sezione** (che costituisce una regione di memoria condivisa); istituito il canale, il client stabilisce se deve inviare messaggi lunghi, in tal caso richiede la creazione di un oggetto sezione. Allo stesso modo, se il server stabilisce che le risposte sono lunghe, provvede alla creazione di un oggetto sezione. Per usare gli oggetti sezione s'invia un breve messaggio contenente un puntatore e le informazioni sulle sue dimensioni. Questo metodo è un po' più complicato del primo, ma evita la copiatura dei dati. In entrambi i casi, se né il client né il server possono rispondere immediatamente alla richiesta, si può impiegare un meccanismo di *callback*; tale meccanismo consente di gestire i messaggi in modo asincrono. La struttura delle chiamate di procedura locali in Windows XP è illustrata nella Figura 3.17.

È importante notare che il meccanismo LPC di Windows XP non è parte della API Win32: quindi, non è accessibile ai programmatori di applicazioni. Le applicazioni Win32 devono comunque usare chiamate di procedura remota; nel caso in cui la chiamata si riferisca a un processo residente sulla stessa macchina del chiamante, il sistema la implementa tramite una chiamata locale. Il meccanismo LPC è anche usato per implementare qualche altra funzione della API Win32.

3.6 Comunicazione nei sistemi client-server

Nel Paragrafo 3.4 ci siamo soffermati su come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione tra sistemi client/server (Paragrafo 1.12.2). Consideriamo qui altre tre strategie: socket, chiamate di procedura remote (RPC) e invocazione di metodi remoti (RMI).

3.6.1 Socket

Una *socket* è definita come l'estremità di un canale di comunicazione. Una coppia di processi che comunica attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo IP concatenato a un numero di porta. In generale, le

socket impiegano un'architettura client-server; il server attende le richieste dei client, stando in ascolto a una porta specificata; quando il server riceve una richiesta, se accetta la connessione proveniente dalla socket del client, si stabilisce la comunicazione. I server che svolgono servizi specifici (come telnet, ftp, e http) stanno in ascolto a porte note (i server telnet alla porta 23, i server ftp alla porta 21, e i server Web o http alla porta 80). Tutte le porte al di sotto del valore 1024 sono considerate note e si usano per realizzare servizi standard.

Quando un processo client richiede una connessione, il calcolatore che lo esegue assegna una porta specifica, che consiste di un numero arbitrario maggiore di 1024. Si supponga per esempio che un processo client presente nel calcolatore `X` con indirizzo IP 146.86.5.20 voglia stabilire una connessione con un server Web (in ascolto alla porta 80) all'indirizzo 161.25.19.8; il calcolatore `X` potrebbe assegnare al client, per esempio, la porta 1625. La connessione sarebbe composta di una coppia di socket: (146.86.5.20:1625) nel calcolatore `X` e (161.25.19.8:80) nel server Web. La Figura 3.18 mostra questa situazione. La consegna dei pacchetti al processo giusto avviene secondo il numero della porta di destinazione.

Tutte le connessioni devono essere uniche; quindi, se un altro processo, nel calcolatore *x*, vuole stabilire un'altra connessione con lo stesso server Web, riceve un numero di porta maggiore di 1024 e diverso da 1625. Ciò assicura che ciascuna connessione sia identificata da una distinta coppia di socket.

Sebbene la maggior parte degli esempi di programmazione di questo testo sia scritta in C, le socket sono illustrate usando il linguaggio Java, poiché offre un'interfaccia alle socket più semplice e dispone di una ricca libreria di strumenti di rete. Il lettore interessato alla programmazione con le socket in C o C++ può consultare le note bibliografiche alla fine del capitolo.

Il linguaggio Java prevede tre tipi differenti di socket: quelle **orientate alla connessione** (TCP) sono realizzate con la classe `Socket`; quelle **prive di connessione** (UDP) usano la classe `DatagramSocket`; il terzo tipo di socket è basato sulla classe `MulticastSocket`; si tratta di una sottoclasse della classe `DatagramSocket` che permette l'invio simultaneo dei dati a diversi destinatari (*multicast*).

Si consideri un esempio in cui un server usa socket TCP orientate alla connessione per fornire l'ora e la data correnti ai client. Il server si pone in ascolto alla porta 6013 – questo intero è arbitrario, purché maggiore di 1024. Quando giunge una richiesta di connessione, il server restituisce la data e l'ora al client.

Il codice del server è mostrato nella Figura 3.19. Il server crea una `ServerSocket` che ascolta alla porta 6013. Il server si pone in ascolto tramite la chiamata bloccante `accept()`, e rimane in attesa fino all'arrivo della richiesta di un client. A quel punto, `accept()` restituisce il numero della socket che il server può usare per comunicare con il client.

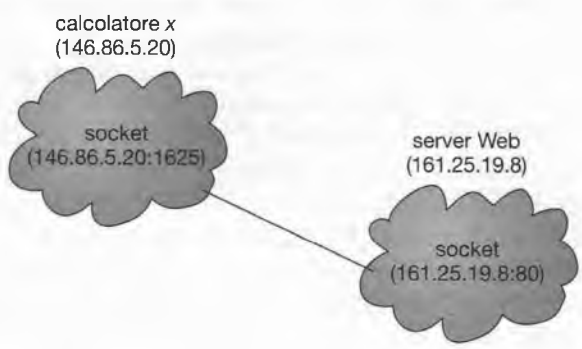

Figura 3.18 Comunicazione tramite socket.

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      // si pone in ascolto di richieste di connessione
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        // scrive la Data sulla socket
        pout.println(new java.util.Date().toString());

        // chiude la socket e ritorna in ascolto
        // di nuove richieste
        client.close();
      }
  }
  catch (IOException ioe) {
    System.err.println(ioe);
  }
 }
}
```

Figura 3.19 Server che comunica ai client la data corrente.

Ecco alcuni dettagli relativi all'uso da parte del server della socket connessa al client. Il server crea da principio un oggetto di classe `PrintWriter` che gli permette di scrivere sulla socket tramite i metodi `print()` e `println()`. Esso manda quindi la data e l'ora al client scrivendo sulla socket tramite `println()`; a questo punto, chiude la socket di comunicazione con il client e torna in attesa di nuove richieste.

Un client comunica con il server creando una socket e collegandosi per suo tramite alla porta su cui il server è in ascolto. L'implementazione è mostrata nella Figura 3.20. Il client crea una `Socket` e richiede una connessione al server alla porta 6013 dell'indirizzo IP 127.0.0.1. Stabilita la connessione, il client può leggere dalla socket tramite le ordinarie istruzioni di I/O. Dopo aver ricevuto la data dal server, il client chiude la socket e termina. L'indirizzo IP 127.0.0.1, noto come **loopback**, è peculiare: è usato da una macchina per riferirsi a se stessa. Tramite questo stratagemma, un client e un server residenti sulla stessa macchina sono in grado di comunicare tramite il protocollo TCP/IP. Nell'esempio, si può sostituire l'indirizzo loopback con l'indirizzo IP di una qualunque macchina che ospiti il server della Figura 3.19. È anche possibile usare un nome simbolico, come www.westminstercollege.edu.

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      //si collega alla porta su cui ascolta il server
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      // legge la data dalla socket
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      // chiude la connessione tramite socket
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Figura 3.20 Client che si collega al server nella Figura 3.19.

La comunicazione tramite socket è diffusa ed efficiente, ma è considerata una forma di comunicazione fra sistemi distribuiti a basso livello. Infatti, le socket permettono unicamente la trasmissione di un flusso non strutturato di byte: è responsabilità del client e del server interpretare e organizzare i dati in forme più complesse. Nei due paragrafi successivi sono illustrati due metodi di comunicazione di più alto livello, le chiamate di procedure remote (RPC) e le invocazioni di metodi remoti (RMI).

3.6.2 Chiamate di procedure remote

Uno tra i più diffusi tipi di servizio remoto è il paradigma della RPC, presentato brevemente nel Paragrafo 3.5.2. La RPC è stata progettata per astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete. Per molti aspetti è simile al meccanismo IPC descritto nel Paragrafo 3.4, ed è generalmente costruita su un sistema di questo tipo. Poiché in un sistema distribuito si eseguono i processi su sistemi distinti, per offrire un servizio remoto occorre impiegare uno schema di comunicazione basato sullo scambio di messaggi. Contrariamente a quel che accade nella funzione IPC, i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone di RPC, in ascolto a una porta del sistema remoto, e contengono un identificatore della

funzione da eseguire e i parametri da inviare a tale funzione. Nel sistema remoto si esegue questa funzione e s'invia ogni risultato al richiedente in un messaggio distinto.

La *porta* è semplicemente un numero inserito all'inizio dei pacchetti di messaggi. Mentre un singolo sistema ha normalmente un solo indirizzo di rete, all'interno dell'indirizzo può avere molte porte che servono a distinguere i numerosi servizi di rete che può fornire. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente; per esempio, per permettere ad altri di ottenere un elenco dei suoi attuali utenti, un sistema deve possedere un demone in ascolto a una porta, per esempio la porta 3027, che realizzi una siffatta RPC. Qualsiasi sistema remoto può ottenere l'informazione richiesta, vale a dire l'elenco degli utenti, inviando un messaggio RPC alla porta 3027 del server; i dati si ricevono in un messaggio di risposta.

La semantica delle RPC permette a un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Il sistema delle RPC nasconde i dettagli necessari che consentono la comunicazione, assegnando un **segmento di codice di riferimento** (*stub*) alla parte client. Esiste in genere un segmento di codice di riferimento per ogni diversa procedura remota. Quando il client la invoca, il sistema delle RPC richiama l'appropriato segmento di codice di riferimento, passando i parametri della procedura remota. Il segmento di codice di riferimento individua la porta del server e struttura i parametri; la strutturazione dei parametri (*marshalling*) implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Il segmento di codice di riferimento quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo segmento di codice di riferimento nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica.

Una questione da affrontare riguarda le differenze nella rappresentazione dei dati nel client e nel server. Si consideri la rappresentazione a 32 bit dei numeri interi; alcuni sistemi, noti come *big-endian*, usano l'indirizzo di memoria maggiore per contenere il byte più significativo; altri, noti come *little-endian*, lo usano per contenere il byte meno significativo. [Questa terminologia deriva dall'analogia di Cohen (Cohen, D., "On Holy Wars and a Plea for Peace", *IEEE Computer Magazine*, vol. 14, pagg. 48-54, ottobre 1981) con la controversia raccontata nei *Viaggi di Gulliver*, Capitolo IV, riguardante l'estremità – larga o stretta – da cui si dovessero rompere le uova per berle.] Per risolvere questo problema, molti sistemi di RPC definiscono una rappresentazione dei dati indipendente dalla macchina. Uno di questi sistemi di rappresentazione è noto come **rappresentazione esterna dei dati** (*external data representation*, XDR). Nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato XDR; nel server, i dati nel formato XDR si convertono nel formato della macchina server.

Un'altra questione importante riguarda la semantica delle chiamate. Infatti, mentre le chiamate locali falliscono in circostanze estreme, le RPC possono non riuscire, o risultare duplicate e dunque eseguite più volte, semplicemente a causa di comuni errori della rete. Un modo per affrontare il problema è di far sì che il sistema operativo agisca sui messaggi *esattamente una volta*, e non *al massimo una volta*. Questo tipo di semantica è comune per le chiamate locali, ma è più difficile da implementare per le RPC.

Si consideri prima la semantica "al massimo una volta". Essa può essere implementata marcando ogni messaggio con la sua ora di emissione (*timestamp*). Il server dovrà mantenere l'archivio di tutti gli orari di emissione dei messaggi già ricevuti e trattati, o perlomeno un archivio che sia abbastanza ampio da identificare i messaggi duplicati. I messaggi in entrata con orario di emissione già presente nell'archivio sono ignorati. I client avranno allora la si-

curezza che la procedura remota sarà eseguita al massimo una volta, anche nel caso di un invio di più copie di uno stesso messaggio. (La generazione dell'orario di emissione è analizzata nel Paragrafo 18.1.)

Per implementare la semantica "esattamente una volta", occorre eliminare il rischio che il server non riceva mai la richiesta. Per ottenere questo risultato, il server deve implementare la semantica "al massimo una volta", ma integrarla con l'invio al client di una ricevuta che attesti l'avvenuta esecuzione della procedura. Ricevute di questo tipo sono molto diffuse nella comunicazione tramite reti. Il client dovrà inviare periodicamente la richiesta RPC finché non ottenga la relativa ricevuta.

Un altro argomento importante riguarda la comunicazione tra server e client. Con le ordinarie chiamate di procedure, durante la fase di collegamento, caricamento o esecuzione di un programma (Capitolo 8), ha luogo una forma di associazione che sostituisce il nome della procedura chiamata con l'indirizzo di memoria della procedura stessa. Lo schema delle RPC richiede una corrispondenza di questo genere tra il client e la porta del server; esiste tuttavia il problema del riconoscimento dei numeri delle porte del server da parte del client. Nessun sistema dispone d'informazioni complete sugli altri sistemi, poiché essi non condividono memoria.

Per risolvere questo problema s'impiegano per lo più due metodi. Con il primo, l'informazione sulla corrispondenza tra il client e la porta del server si può predeterminare fissando gli indirizzi delle porte: una RPC si associa nella fase di compilazione a un numero di porta fisso; il server non può modificare il numero di porta del servizio richiesto. Con il secondo metodo la corrispondenza si può effettuare dinamicamente tramite un meccanismo di *rendezvous*. Generalmente il sistema operativo fornisce un demone di rendezvous (*matchmaker*) a una porta di RPC fissata. Un client invia un messaggio, contenente il nome della RPC, al demone di rendezvous per richiedere l'indirizzo della porta della RPC da eseguire. Il demone risponde col numero di porta, e la richiesta d'esecuzione della RPC si può inviare a quella porta fino al termine del processo (o fino alla caduta del server). Questo metodo richiede un ulteriore carico a causa della richiesta iniziale, ma è più flessibile del primo metodo. La Figura 3.21 illustra un esempio d'interazione.

Lo schema della RPC è utile nella realizzazione di un file system distribuito (Capitolo 17); un sistema di questo tipo si può realizzare come un insieme di demoni e client di RPC. I messaggi s'indirizzano alla porta del file system distribuito su un server in cui deve avvenire l'operazione sui file. I messaggi contengono le operazioni da svolgere nei dischi: `read`, `write`, `rename`, `delete` o `status`, corrispondenti alle normali chiamate di sistema che si usano per i file. Il messaggio di risposta contiene i dati risultanti da quella chiamata, che il demone del DFS esegue su incarico del client. Un messaggio può, per esempio, contenere una richiesta di trasferimento di un intero file a un client, oppure semplici richieste di blocchi. Nel secondo caso per trasferire un intero file possono essere necessarie parecchie richieste di questo tipo.

3.6.3 Pipe

Una **pipe** agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi (IPC) nei sistemi UNIX e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro, sebbene con qualche limitazione. Quando si implementa una pipe devono essere prese in considerazione quattro questioni.

1. La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?

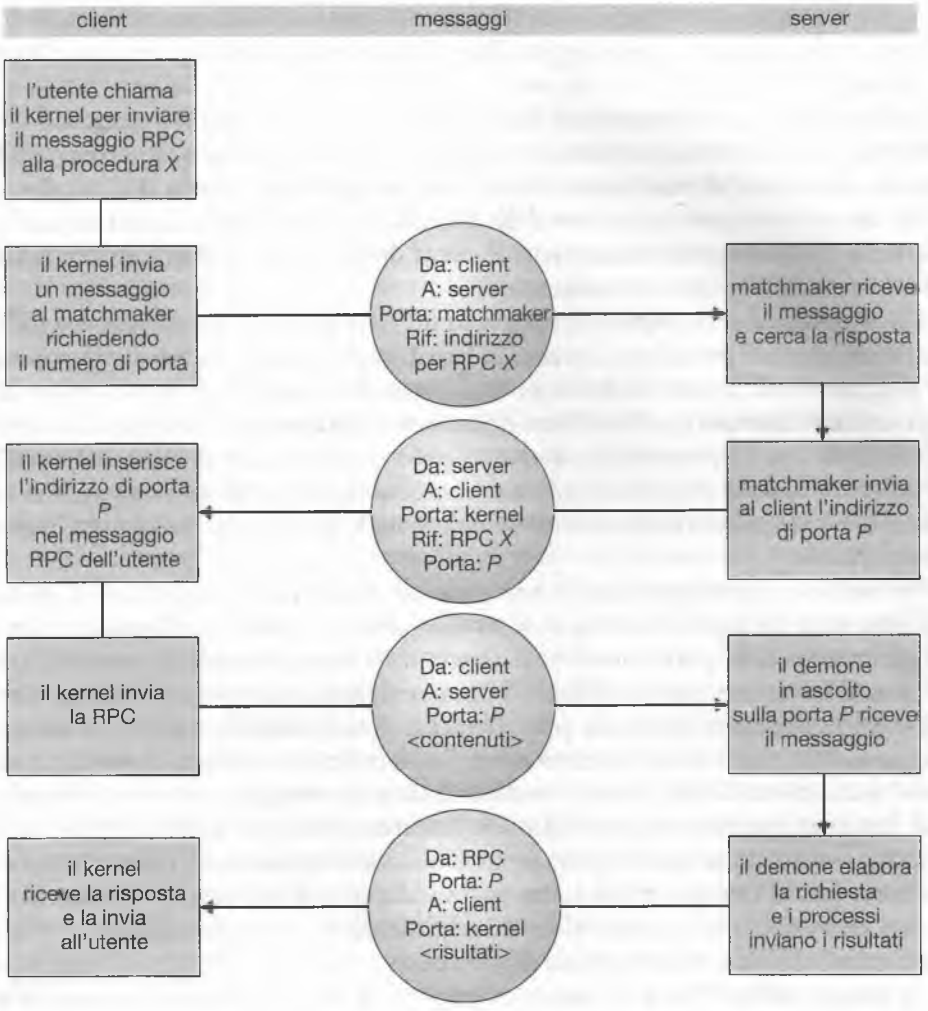

Figura 3.21 Esecuzione di una chiamata di procedura remota (RPC).

2. Se è ammessa la comunicazione a doppio senso, essa è di tipo *half duplex* (i dati possono viaggiare in un'unica direzione alla volta) o *full duplex* (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
3. Deve esistere una relazione (del tipo *padre-figlio*) tra i processi in comunicazione?
4. Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

Nei paragrafi seguenti esploriamo due tipi comuni di pipe utilizzate sia in UNIX sia in Windows.

3.6.3.1 Pipe convenzionali

Le pipe convenzionali permettono a due processi di comunicare secondo una modalità standard chiamata del produttore-consumatore. Il produttore scrive a una estremità del canale (l'**estremità dedicata alla scrittura**, o *write-end*) mentre il consumatore legge dall'altra estremità (l'**estremità dedicata alla lettura**, o *read-end*). Le pipe convenzionali sono quindi unidirezionali, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la

comunicazione a doppio senso devono essere utilizzate due *pipe*, ognuna delle quali manda i dati in una differente direzione. Illustreremo la costruzione di pipe convenzionali sia in UNIX sia in Windows. In entrambi i programmi di esempio un processo scrive sulla pipe il messaggio `Greetings`, mentre l'altro lo legge dall'altra estremità della pipe.

Nel sistema UNIX le pipe convenzionali sono costruite utilizzando la funzione

```
pipe(int fd[])
```

Essa crea una pipe alla quale si può accedere tramite i descrittori del file `int fd[]:fd[0]` è l'estremità dedicata alla lettura, mentre `fd[1]` è l'estremità dedicata alla scrittura. Il sistema UNIX considera una pipe come un tipo speciale di file; si può così accedere alle pipe tramite le usuali chiamate di sistema `read()` e `write()`.

Non si può accedere a una pipe al di fuori del processo che la crea. Solitamente un processo padre crea una pipe e la utilizza per comunicare con un processo figlio generato con il comando `fork()`. Come già indicato nel Paragrafo 3.3.1, il processo figlio eredita i file aperti dal processo padre. Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre. La Figura 3.22 illustra la relazione del descrittore di file `fd` rispetto ai processi padre e figlio.

Nel programma UNIX mostrato nella Figura 3.23 (che continua nella Figura 3.24) il processo padre crea una pipe e in seguito esegue una chiamata `fork()`, generando un pro-

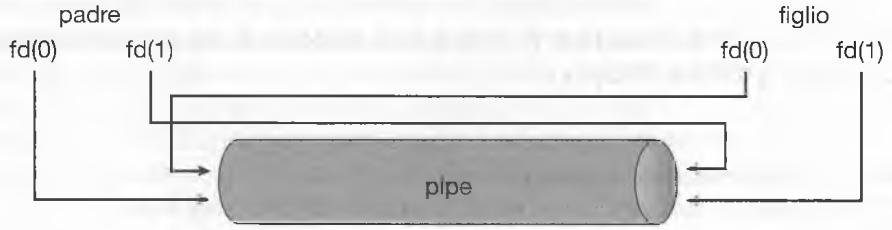

Figura 3.22 Descrittori di file per una pipe convenzionale.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER SIZE 25
#define READ END 0
#define WRITE END 1

int main(void)
{
char write msg[BUFFER SIZE] = "Greetings";
char read msg[BUFFER SIZE];
int fd[2];
pid t pid;
```

Il programma continua nella Figura 3.24

Figura 3.23 Pipe convenzionali in UNIX.

```
  /* crea la pipe */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }

  /* crea tramite fork un processo figlio */
  pid = fork();

  if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
  }

  if (pid > 0) { /* processo padre */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[READ END]);

    /* scrive sulla pipe */
    write(fd[WRITE END], write msg, strlen(write msg)+1);

    /* chiude l'estremità della pipe dedicata alla scrittura */
    close(fd[WRITE END]);

  }
  else { /* processo figlio */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[WRITE END]);

    /* legge dalla pipe */
    read(fd[READ END], read msg, BUFFER SIZE);
    printf("read %s",read msg);

    /* chiude l'estremità della pipe dedicata alla lettura */
    close(fd[READ END]);
  }

  return 0;
}
```

Figura 3.24 Continuazione del programma della Figura 3.23.

cesso figlio. Ciò che succede dopo la chiamata `fork()` dipende da come i dati fluiscono nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale. Sebbene il programma mostrato nella Figura 3.23 non richieda questa azione è importante assicurare che un processo che legge dalla pipe possa rilevare il ca-

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS INFORMATION pi;
char message[BUFFER SIZE] = "Greetings";
DWORD written;
```

Il programma continua nella Figura 3.26

Figura 3.25 Pipe anonime in Windows (processo padre).

rattere di terminazione EOF (`read()` restituisce 0) quando chi scrive ha chiuso la sua estremità della pipe.

Nei sistemi Windows le pipe convenzionali sono denominate **pipe anonime** e si comportano analogamente alle loro equivalenti in UNIX: sono unidirezionali e utilizzano relazioni del tipo padre-figlio tra i processi coinvolti nella comunicazione. Inoltre, l'attività di lettura e scrittura sulla pipe può essere eseguita con le usuali funzioni `ReadFile()` e `WriteFile()`. L'API Win32 per creare le pipe è la funzione `CreatePipe()`, che riceve in ingresso quattro parametri: (1) handle separati per leggere e (2) scrivere sulla pipe, oltre a (3) una istanza della struttura `STARTUPINFO`, usata per specificare che il processo figlio erediti gli handle della pipe. Inoltre, può essere specificata (4) la dimensione della pipe (in byte).

La Figura 3.25 (e la continuazione nella Figura 3.26) illustrano un processo padre che crea una pipe anonima per comunicare con il proprio figlio. A differenza dei sistemi UNIX, dove un processo figlio eredita automaticamente una pipe dal proprio padre, Windows richiede al programmatore di specificare quali attributi saranno ereditati dal processo figlio. Ciò avviene innanzitutto inizializzando la struttura SECURITY_ATTRIBUTES per permettere che gli handle siano ereditati e poi reindirizzando lo standard input o lo standard output del processo figlio verso l'handle di scrittura o di lettura della pipe, rispettivamente. Dato che il figlio leggerà dalla pipe, il padre deve reindirizzare lo standard input del figlio verso l'handle di lettura della pipe stessa. Inoltre, dal momento che le pipe sono half duplex, è necessario proibire al figlio di ereditare l'handle di scrittura della pipe. La creazione del processo figlio avviene come nel programma nella Figura 3.12, fatta eccezione per il quinto parametro che in questo caso viene impostato al valore TRUE per indicare che il processo figlio eredita degli handle specifici dal proprio padre. Prima di scrivere sulla pipe, il padre ne chiude l'estremità di lettura, che è inutilizzata. Il processo figlio che legge dalla pipe è mostrato nella Figura 3.27. Prima di leggere dalla pipe, questo programma ottiene l'handle di lettura invocando `GetStdHandle()`.

Si noti bene che le pipe convenzionali richiedono una relazione di parentela padre-figlio tra i processi comunicanti, sia in UNIX sia in Windows. Ciò significa che queste pipe possono essere utilizzate soltanto per la comunicazione tra processi in esecuzione sulla stessa macchina.

```
/*imposta gli attributi di sicurezza in modo che le pipe siano
  ereditate */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* alloca la memoria */
ZeroMemory(&pi, sizeof(pi));

/* crea la pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
  fprintf(stderr, "Create Pipe Failed");
  return 1;
}

/* prepara la struttura START_INFO per il processo figlio */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* reindirizza lo standard input verso l'estremità della pipe
   dedicata alla lettura */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* non permette al processo figlio di ereditare l'estremità
   della pipe dedicata alla scrittura */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* crea il processo figlio */
CreateProcess(NULL, "child.exe", NULL, NULL,
  TRUE, /* inherit handles */
  0, NULL, NULL, &si, &pi);

/* chiude l'estremità inutilizzata della pipe */
CloseHandle(ReadHandle);

/* il padre scrive sulla pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
  fprintf(stderr, "Error writing to pipe.");

/* chiude l'estremità della pipe dedicata alla scrittura */
CloseHandle(WriteHandle);

/* attende la terminazione del processo figlio */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figura 3.26 Continuazione del programma nella Figura 3.25.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

  /* riceve l'handle di lettura della pipe */
  ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

  /* il figlio legge dalla pipe */
  if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
    printf("child read %s",buffer);
  else
    fprintf(stderr, "Error reading from pipe");
  return 0;
}
```

Figura 3.27 Pipe anonime in Windows (processo figlio).

3.6.3.2 Named pipe

Le pipe convenzionali offrono un meccanismo semplice di comunicazione tra una coppia di processi. Tuttavia, le pipe convenzionali esistono solo mentre i processi stanno comunicando. Sia in UNIX sia in Windows, una volta che i processi hanno terminato di comunicare, le pipe convenzionali cessano di esistere.

Le **named pipe** costituiscono uno strumento di comunicazione molto più potente; la comunicazione può essere bidirezionale, e la relazione di parentela padre-figlio non è necessaria. Una volta che si sia creata la named pipe, diversi processi possono utilizzarla per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori. In più, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. Sia UNIX sia Windows mettono a disposizione le named pipe, nonostante ci siano grandi differenze nei dettagli dell'implentazione. Di seguito, prendiamo in esame le named pipe in ciascuno dei due sistemi.

Nei sistemi UNIX le named pipe sono dette FIFO. Una volta create, esse appaiono come normali file all'interno del file system. Una FIFO viene creata mediante una chiamata di sistema `mkfifo()` e viene poi manipolata con le usuali chiamate di sistema `open()`, `read()`, `write()` e `close()`; essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system. Nonostante i file FIFO permettano la comunicazione bidirezionale, l'unica tipologia di trasmissione consentita è quella half duplex. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due FIFO. Per utilizzare le FIFO i processi comunicanti devono risiedere sulla stessa macchina: se è richiesta la comunicazione tra più macchine devono essere impiegate le socket (si veda il Paragrafo 3.6.1).

Rispetto alle loro controparti in UNIX, le named pipe su un sistema Windows offrono un meccanismo di comunicazione più ricco. È permessa la comunicazione full duplex e i

LE PIPE IN PRATICA

Le pipe sono usate abbastanza spesso dalla riga di comando di UNIX in situazioni nelle quali l'output di un comando serve da input per un secondo comando. Ad esempio, il comando `ls` di UNIX elenca il contenuto di una directory. Per directory particolarmente grandi, l'output può scorrere su diverse schermate. Il comando `more` gestisce l'output mostrando solo una schermata alla volta; l'utente deve premere la barra spaziatrice per muoversi da una schermata all'altra. Istituendo una pipe tra i comandi `ls` e `more` (in esecuzione come singoli processi) si fa in modo che l'output di `ls` venga inviato all'input di `more`, permettendo così all'utente di vedere il contenuto di una grande directory una schermata alla volta. Dalla riga di comando si può costruire una pipe utilizzando il carattere |. Il comando completo è quindi

```
ls|more
```

In questo scenario, il comando `ls` funge da produttore, e il suo output è consumato dal comando `more`.

I sistemi Windows offrono un comando `more` per la shell DOS con una funzionalità analoga al corrispettivo di UNIX. Anche la shell DOS utilizza il carattere | per creare una pipe. L'unica differenza è costituita dal fatto che, per restituire il contenuto di una directory, DOS utilizza il comando `dir` anziché `ls`. Il comando equivalente in DOS è quindi

```
dir|more
```

processi comunicanti possono risiedere sia sulla stessa macchina sia su macchine diverse. Inoltre, attraverso una FIFO di UNIX possono essere trasmessi solo dati byte-oriented, mentre i sistemi Windows permettono la trasmissione di dati sia byte-oriented sia message-oriented. Le named pipe vengono create con la funzione `CreateNamedPipe()` e un client può connettersi a una named pipe tramite `ConnectNamedPipe()`. La comunicazione attraverso le named pipe avviene grazie alle funzioni `ReadFile()` e `WriteFile()`.

3.7 Sommario

Un processo è un programma in esecuzione. Nel corso delle sue attività, un processo cambia stato, e tale stato è definito dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati: nuovo, pronto, esecuzione, attesa o arresto. In un sistema operativo ogni processo è rappresentato dal proprio blocco di controllo del processo (PCB).

Un processo, quando non è in esecuzione, è inserito in una coda d'attesa. Le due classi principali di code in un sistema operativo sono le code di richieste di I/O e la coda dei processi pronti per l'esecuzione, quest'ultima contenente tutti i processi pronti per l'esecuzione che si trovino in attesa della CPU. Ogni processo è rappresentato da un PCB; i PCB si possono collegare tra loro in modo da formare una coda dei processi pronti. Lo scheduling a lungo termine (o *job scheduling*) consiste nella scelta dei processi che si contenderanno la CPU. Normalmente lo scheduling a lungo termine è influenzato in modo consistente da considerazioni riguardanti l'assegnazione delle risorse, in particolar modo quelle concernenti la gestione della memoria. Lo scheduling a breve termine (o scheduling della CPU) consiste nella selezione di un processo dalla coda dei processi pronti.

I sistemi operativi devono implementare un meccanismo per generare processi figli da un processo genitore. Genitore e figli possono girare in concomitanza, oppure il genitore po-

trà essere posto in attesa della terminazione dei figli. L'esecuzione concorrente è ampiamente giustificabile dalla necessità di condividere informazioni e dal potenziale aumento della velocità di calcolo, oltre che da considerazioni legate alla modularità e alla convenienza.

I processi in esecuzione nel sistema operativo possono essere indipendenti o cooperanti. I processi cooperanti devono avere i mezzi per comunicare tra loro. Fondamentalmente esistono due schemi complementari di comunicazione: memoria condivisa e scambio di messaggi. Nel metodo con memoria condivisa i processi in comunicazione devono condividere alcune variabili, per mezzo di cui i processi scambiano informazioni; il compito della comunicazione è lasciato ai programmatori di applicazioni; il sistema operativo deve semplicemente offrire la memoria condivisa. Il metodo con scambio di messaggi permette di compiere uno scambio di messaggi tra i processi; in questo caso il compito di attuare la comunicazione è del sistema operativo. Questi due schemi non sono mutuamente esclusivi, e si possono impiegare insieme in uno stesso sistema.

La comunicazione nei sistemi client/server può impiegare (1) socket, (2) chiamate di procedure remote (RPC) o (3) chiamate di metodi remoti (RMI). Una connessione tra una coppia di applicazioni consiste di una coppia di socket, ciascuna a un'estremità del canale di comunicazione. Le RPC sono un'altra forma di comunicazione distribuita: una RPC si verifica quando un processo (o un thread) invoca una procedura in un'applicazione remota. Le RMI sono la versione del linguaggio Java delle RPC; consentono a un thread di invocare un metodo su un oggetto remoto esattamente come se si trattasse di oggetto locale. La differenza principale tra le RPC e le RMI consiste nel fatto che i dati passati a una procedura remota hanno la forma di un'ordinaria struttura dati, mentre le RMI consentono anche il passaggio di oggetti.

Esercizi pratici

3.1 Palm OS non offre strumenti per gestire processi concorrenti. Esponete tre principali complicazioni che la gestione di processi concorrenti crea al sistema operativo.

3.2 Il processore Sun UltraSPARC ha diversi insiemi di registri. Descrivete ciò che avviene quando si ha un cambio di contesto nel caso in cui il contesto successivo sia già caricato in un determinato insieme di registri. Che cosa succede se il contesto successivo è nella memoria (invece che nei registri) e tutti i registri sono in uso?

3.3 Quando un processo crea un nuovo processo utilizzando l'istruzione `fork()`, quale dei seguenti stati è condiviso tra il processo padre e il processo figlio?

a. Stack.
b. Heap.
c. Segmenti di memoria condivisi.

3.4 A proposito del meccanismo RPC, considerate la semantica "exactly once". L'algoritmo che implementa questa semantica funziona correttamente anche se il messaggio ACK che restituisce al client va perso a causa di un problema di rete? Descrivete la sequenza di messaggi scambiati e prendete nota di quando la semantica "exactly once" viene ancora preservata.

3.5 Assumendo che un sistema distribuito sia sensibile a malfunzionamento del server, quali meccanismi sarebbero richiesti per garantire la semantica "exactly once" per l'esecuzione di RPC?

3.6 Descrivete le differenze tra scheduling a breve termine, a medio termine e a lungo termine.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  /* crea mediante fork un processo figlio */
  fork();

  /* crea un altro processo figlio */
  fork();

  /* e ne crea un altro ancora */
  fork();

  return 0;
}
```

Figura 3.28 Quanti processi vengono creati?

3.7 Descrivete le azioni intraprese dal kernel nell'esecuzione di un cambio di contesto tra processi.

3.8 Costruite un albero di processi simile a quello nella Figura 3.9. Utilizzate il comando `ps -ael` per ottenere informazioni sui processi in UNIX o in Linux. Utilizzate il comando `man ps` per ottenere più informazioni sul comando `ps`. Per i sistemi Windows, dovrete usare il task manager.

3.9 Considerando anche il processo padre iniziale, quanti processi vengono creati dal programma della Figura 3.28?

3.10 In riferimento al programma nella Figura 3.29, identificate i valori dei `pid` alle linee `A`, `B`, `C` e `D`. (Assumete che i `pid` attuali del padre e del figlio siano rispettivamente 2600 e 2603). Si veda la Figura 3.30.

3.11 Fornite un esempio di situazione nella quale le pipe convenzionali siano più adatte delle named pipe e un esempio di situazione nella quale le named pipe siano invece più indicate delle pipe convenzionali.

3.12 In riferimento al meccanismo RPC, descrivete i possibili effetti negativi della mancata implementazione della semantiche "al massimo una volta" o di quella "esattamene una volta". Considerate una possibile applicazione di un tale meccanismo che non le implementi.

3.13 Descrivete l'output del programma nella Figura 3.30 alla `RIGA A`.

3.14 Analizzate vantaggi e svantaggi delle tecniche elencate di seguito, considerando sia il punto di vista del sistema sia quello del programmatore.

a. Comunicazione sincrona e asincrona.
b. Gestione automatica o esplicita del buffer.
c. Trasmissione per copia e trasmissione per riferimento.
d. Messaggi a lunghezza fissa e variabile.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid t pid, pid1;

  /* crea mediante fork un processo figlio */
  pid = fork();

  if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* processo figlio */
    pid1 = getpid();
    printf("child: pid = %d",pid); /* A */
    printf("child: pid1 = %d",pid1); /* B */
  }
  else { /* processo padre */
    pid1 = getpid();
    printf("parent: pid = %d",pid); /* C */
    printf("parent: pid1 = %d",pid1); /* D */
    wait(NULL);
  }

  return 0;
}
```

Figura 3.29 Quali sono i valori dei pid?

Problemi di programmazione

3.15 La successione di Fibonacci comincia con 0, 1, 1, 2, 3, 5, 8, ... La sua definizione ricorsiva è:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Scrivete un programma C che usi la chiamata di sistema `fork()` per generare la successione di Fibonacci all'interno del processo figlio. Il numero di termini da generare sarà specificato dalla riga di comando. Per esempio, se il parametro passato dalla riga di comando fosse 5, il programma dovrebbe far produrre in uscita al processo figlio i primi 5 termini della successione di Fibonacci. Visto che genitore e figlio hanno copie private dei dati, sarà il figlio a dover produrre i dati in uscita. Il genitore dovrà rimanere in attesa tramite `wait()` fino alla terminazione del figlio. Implementate i necessari controlli per garantire che il valore in ingresso dalla riga di comando sia un intero non negativo.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid t pid;

  pid = fork();

  if (pid == 0) { /* processo figlio */
    value += 15;
    return 0;
  }
  else if (pid > 0) { /* processo padre */
    wait(NULL);
    printf("PARENT: value = %d",value); /* Linea A */
    return 0;
  }
}
```

Figura 3.30 Quale sarà l'output della Linea A?

3.16 Riprendete l'esercizio precedente, usando però la API Win32 e la chiamata `CreateProcess()`. Dovrete specificare un programma a se stante come parametro di `CreateProcess()`: esso sarà eseguito in qualità di processo figlio, e dovrà produrre in uscita la successione di Fibonacci. Implementate i necessari controlli per garantire che il valore in ingresso dalla riga di comando sia un intero non negativo.

3.17 Modificate il server nella Figura 3.19 in modo che fornisca aforismi casuali in luogo di data e ora. Gli aforismi possono richiedere più linee di testo. Il client nella Figura 3.20 può essere adattato alla lettura degli aforismi restituiti dal server.

3.18 Un server eco è un server che restituisce ai client esattamente ciò che essi gli inviano. Se un client, per esempio, invia al server la stringa *Ciao*, il server risponderà con gli stessi dati: ossia, invierà al client la stringa *Ciao*. Scrivete un server eco usando la API Java descritta nel Paragrafo 3.6.1. Il server rimarrà in attesa dei client tramite il metodo `accept()`. Dopo aver accettato una connessione, il client eseguirà il ciclo seguente:

- leggerà i dati dalla socket connessa con il client, ponendoli in un buffer;
- scriverà i contenuti del buffer sulla socket connessa con il client.

Il server uscirà dal ciclo solo dopo aver stabilito che il client ha chiuso la connessione. Il server nella Figura 3.19 impiega la classe Java `java.io.BufferedReader`, che estende la classe `java.io.Reader`, usata per leggere flussi di caratteri. Il server eco, però, potrebbe ricevere dati di altro tipo dal client – per esempio dati binari. La classe `java.io.InputStream` tratta i dati byte per byte, e non carattere per carattere. È quindi necessario che il server eco impieghi una classe che estenda `java.io.InputStream`. Il metodo `read()` di `java.io.InputStream` restituisce –1 quando il client ha chiuso la connessione.

3.19 Nell'Esercizio 3.15 è il processo figlio a dover produrre in uscita la successione di Fibonacci, perché processo genitore e figlio possiedono copie private dei dati. Un approccio alternativo per la stesura di questo programma è impiegare un segmento di memoria condiviso tra genitore e figlio. In questo modo, il figlio potrà scrivere i numeri di Fibonacci richiesti in memoria condivisa, e il genitore potrà fornirli all'utente alla terminazione del figlio. A causa della condivisione della memoria, i dati in questione saranno accessibili al genitore anche dopo la terminazione del figlio. Il programma dovrà essere implementato con la API POSIX descritta nel Paragrafo 3.5.1. Per prima cosa, bisogna prevedere una struttura dati per il segmento di memoria condivisa. La soluzione più semplice è impiegare il costrutto C `struct`, dichiarando due elementi al suo interno: (1) un vettore di dimensione fissa `MAX_SEQUENCE` che conterrà i valori della successione; e (2) il numero di termini `sequence_size` che il figlio deve produrre, dove `sequence_size` ≤ `MAX_SEQUENCE`. Ecco il codice relativo:

```
#define MAX_SEQUENCE 10

typedef struct {
  long fib_sequence[MAX_SEQUENCE];
  int sequence_size;
}shared_data;
```

Il processo genitore eseguirà i passi seguenti:

a. accetterà il parametro passato dalla riga di commando, e controllerà che si tratti di un intero non negativo ≤ `MAX_SEQUENCE`;
b. allocherà l'area di memoria condivisa di dimensione `shared_data`;
c. annetterà l'area condivisa al suo spazio degli indirizzi;
d. imposterà il valore di `sequence_size` al parametro di cui al punto a;
e. genererà il processo figlio e invocherà `wait()` per attendere la sua terminazione;
f. scriverà in uscita i termini della successione di Fibonacci presenti in memoria condivisa;
g. eliminerà l'area di memoria condivisa dal suo spazio degli indirizzi e dal sistema.

Il processo figlio, in quanto copia del processo genitore, avrà accesso alla memoria condivisa. Il processo figlio scriverà gli opportuni termini della successione di Fibonacci in memoria condivisa, per poi eliminarla dal proprio spazio degli indirizzi e terminare l'esecuzione.
Uno dei problemi inerenti alla cooperazione fra processi è la loro sincronizzazione. In questo esercizio, il processo genitore non deve cominciare a scrivere i termini della successione in uscita prima della terminazione del figlio. La sincronizzazione è implementata tramite la chiamata `wait()`, che sospende il genitore fino alla terminazione del figlio.

3.20 Utilizzando pipe convenzionali, scrivete un programma nel quale un processo manda una stringa a un secondo processo, il quale cambia le lettere maiuscole del messaggio in minuscole, e viceversa, per poi restituire il risultato al primo processo. Ad esempio, se il primo processo manda il messaggio `Hi There`, il secondo restituirà `hI tHERE`. Ciò richiede l'utilizzo di due pipe, una per mandare il messaggio originale dal primo al secondo processo e l'altra per mandare il messaggio modificato dal secondo processo al primo. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

3.21 Scrivete un programma `FileCopy` per la copia di file, usando le pipe convenzionali. Questo programma riceverà in ingresso due parametri: il primo è il nome del file che deve essere copiato, il secondo è il nome del file copiato. Il programma creerà una pipe convenzionale e scriverà i contenuti del file da copiare nella pipe. Il processo figlio leggerà il file dalla pipe e lo scriverà nel suo file di destinazione. Se ad esempio invochiamo il programma come segue:

```
FileCopy input.txt copy.txt
```

il file `input.txt` sarà scritto sulla pipe. Il processo figlio leggerà i contenuti del file e li scriverà nel file di destinazione `copy.txt`. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

3.22 Molte implementazioni di UNIX e Linux offrono il comando `ipcs` per elencare lo stato dei meccanismi di comunicazione POSIX tra processi, riportando anche informazioni sulle aree di memoria condivisa. Il comando attinge molte delle informazioni dalla struttura `shmid_ds`, contenuta nel file `/usr/include/sys/shm.h`. Alcuni dei campi sono:

- `int shm_segsz` – la dimensione dell'area;
- `short shm_nattch` – numero di annessioni eseguite;
- `struct ipc_perm shm_perm` – struttura contenente i permessi d'accesso all'area.

La struttura, che si trova nel file `/usr/include/sys/ipc.h`, contiene i campi:

- unsigned `short uid` – identificatore utente della memoria condivisa;
- unsigned `short mode` – modalità d'accesso;
- `key_t key` (in Linux, `__key`) – identificatore-chiave specificato dall'utente.

I valori delle modalità d'accesso dipendono dai parametri passati a `shmget()` al momento dell'allocazione della memoria condivisa. La tabella seguente riassume la codifica dei permessi nel campo `unsigned short mode`:

Modalità	Significato
0400	Permesso di lettura per il proprietario.
0200	Permesso di scrittura per il proprietario.
0040	Permesso di lettura per il gruppo.
0020	Permesso di scrittura per il gruppo.
0004	Permesso di lettura per chiunque.
0002	Permesso di scrittura per chiunque.

I permessi possono essere letti usando l'operatore `&`, che esegue un *AND* bit a bit degli operandi. Se, per esempio, l'espressione `mode & 0400` è valutata true, il proprietario del segmento condiviso ha il permesso di leggere dal segmento.
Le aree di memoria condivisa sono individuabili dal valore intero restituito da `shmget()`, o da un identificatore-chiave specificato dall'utente. La struttura `shm_ds` relativa a un dato identificatore intero si può ottenere con la chiamata di sistema seguente:

```
/* identificatore del segmento di memoria condivisa*/
    int segment_id;
    shm_ds shmbuffer;

    shmctl(segment_id, IPC_STAT, &shmbuffer);
```

Se la chiamata ha successo, il valore restituito è zero, altrimenti è –1.
Scrivete un programma C che accetti in ingresso l'identificatore di un segmento di memoria condivisa. Il programma invocherà `shmctl()` per ottenere la struttura `shm_ds`, e stamperà i valori seguenti relativi al segmento associato all'identificatore in ingresso:

- ID del segmento.
- Identificatore-chiave.
- Modo d'accesso.
- UID del proprietario.
- Dimensione.
- Numero di annessioni.

Progetti di programmazione

3.23 Passare messaggi POSIX

Questo progetto consiste nell'usare code di messaggi POSIX per comunicare temperature tra quattro processi esterni e un processo centrale. Il progetto può essere realizzato su sistemi che mettono a disposizione il passaggio di messaggi POSIX, come UNIX, Linux e Mac OS X.

Parte 1: Panoramica

Quattro processi esterni comunicheranno le temperature a un processo centrale, che a sua volta risponderà con la propria temperatura e indicherà se l'intero processo si è stabilizzato. Ogni processo riceverà la propria temperatura iniziale al momento della creazione e ricalcolerà una nuova temperatura secondo le seguenti formule:

nuova temp processo esterno =
*(miaTemp * 3+2 * tempCentrale) / 5*;

nuova temp processo centrale =
*(2 * tempCentrale + quattro temp ricevute dai processi esterni) / 6*;

Inizialmente ciascun processo esterno manderà la propria temperatura alla mailbox del processo centrale. Se le quattro temperature coincidono con quelle spedite dai quattro processi durante l'ultima iterazione, il sistema si è stabilizzato. In questo caso il processo centrale notificherà a ogni processo esterno che il suo compito è terminato (insieme al compito del processo centrale stesso), e ogni processo emetterà la temperatura finale stabilizzata. Se il sistema non si è ancora stabilizzato, il processo centrale manderà la sua nuova temperatura alla mailbox di ognuno dei processi esterni e resterà in attesa delle risposte. I processi continueranno l'esecuzione finché le temperature non si saranno stabilizzate.

Parte 2: Sistema di passaggio dei messaggi

I processi possono scambiare messaggi usando quattro chiamate di sistema: `msgget()`, `msgsnd()`, `msgrcv()` e `msgctl()`. La funzione `msgget()` converte il nome di una mailbox in un identificatore numerico di una coda di messaggi, chiamato `msqid`. (Il nome di

una mailbox è il nome di una coda di messaggi condivisa tra i processi cooperanti). L'identificatore interno restituito da `msgget()`, `msqid`, deve essere passato a tutte le successive chiamate di sistema che utilizzano la coda di messaggi per realizzare la comunicazione tra processi. L'invocazione tipica di `msgget()` è la seguente:

```
msqid = msgget(1234, 0600 | IPC_CREAT);
```

Il primo parametro è il nome della mailbox; il secondo parametro istruisce il sistema operativo affinché crei la coda di messaggi se ancora non esiste, con privilegi di lettura e scrittura solo per processi con lo stesso user id del processo corrente. Se esiste già una coda di messaggi associata al nome di questa mailbox, `msgget()` restituisce lo `msqid` della mailbox esistente. Per evitare di collegarsi a una coda di messaggi preesistente, un processo può prima provare a collegarsi alla mailbox omettendo `IPC_CREAT` e poi controllare il valore restituito da `msgget()`. Se `msqid` è negativo, si è verificato un errore durante la chiamata di sistema, e può essere consultata la variabile ad accesso globale `errno` per determinare se l'errore è dovuto all'esistenza della coda di messaggi, oppure ad altre ragioni. Se il processo determina che la mailbox non esiste, la può creare, includendo `IPC_CREAT` nella chiamata. (Per questo progetto, la strategia appena descritta non dovrebbe essere necessaria se gli studenti lavorano su PC indipendenti o se utilizzano delle varianti specifiche di nomi di mailbox assegnate dal docente).

Una volta ottenuto uno `msqid` valido, un processo può cominciare a utilizzare `msgsnd()` per inviare messaggi e `msgrcv()` per riceverli. I messaggi hanno un formato simile a quello descritto nel Paragrafo 3.5.2: sono costituiti da una porzione a lunghezza fissa iniziale, seguita da una porzione a lunghezza variabile. Ovviamente, il mittente e il destinatario devono concordare un formato per i messaggi che intendono scambiarsi. Visto che il sistema operativo specifica un campo della porzione a lunghezza fissa di ciascun messaggio, e poiché almeno un'informazione sarà inviata al processo destinatario, è logico creare un'aggregazione di dati per ogni tipo di messaggio utilizzando una `struct`. Il primo campo di ciascuna `struct` deve essere un `long`, e contiene la priorità del messaggio. (Per questo progetto le priorità dei messaggi sono irrilevanti; raccomandiamo quindi che il primo campo di ciascun messaggio abbia sempre lo stesso valore intero, ad esempio 2). Gli altri campi dei messaggi contengono le informazioni da condividere tra i processi comunicanti. Sono necessari tre campi aggiuntivi: (1) la temperatura da comunicare, (2) il numero di processo del processo esterno che invia il messaggio (0 per il processo centrale) e (3) un flag impostato a 0, che il processo centrale imposterà a 1 quando il sistema diverrà stabile. Ecco una `struct` del tipo appena descritto:

```
struct {
      long priority;
      int temp;
      int pid;
      int stable;
} msgp;
```

Assumendo che `msqid` sia già stato creato, un possibile utilizzo di `msgsnd()` e `msgrcv()` è il seguente:

```
int stat, msqid;

stat = msgsnd(msqid, &msgp,
              sizeof(msgp)-sizeof(long), 0);

stat = msgrcv(msqid, &msgp,
              sizeof(msgp)-sizeof(long), 2, 0);
```

Il primo parametro in entrambe le chiamate di sistema deve essere uno `msqid` valido, altrimenti è restituito un valore negativo. (Entrambe le funzioni restituiscono il numero di byte inviati o ricevuti una volta che l'operazione è terminata con successo). Il secondo parametro è l'indirizzo dove trovare o memorizzare il messaggio da inviare o ricevere, seguito dal numero dei byte da inviare o ricevere. Il parametro finale 0 indica che le operazioni saranno sincrone e che il mittente si bloccherà se la coda di messaggi è piena. (Nel caso in cui fossero richieste operazioni asincrone, o se non si volesse il blocco del mittente, si utilizzerebbe `IPC_NOWAIT`. Ogni singola coda di messaggi può contenere un numero massimo di messaggi o di byte. È dunque possibile che la coda si riempia, ragion per cui un mittente può bloccarsi mentre tenta di trasmettere un messaggio). Il 2 che compare prima di questo parametro finale in `msgrcv()` indica il livello minimo di priorità dei messaggi che il processo desidera ricevere. Se si tratta di un'operazione sincrona, il destinatario aspetterà finché un messaggio di quella priorità (o di priorità maggiore) raggiunga la mailbox `msqid`.

Quando un processo ha terminato di utilizzare una coda di messaggi, questa deve essere rimossa, in modo che la mailbox possa essere sfruttata anche da altri processi. A meno che non venga rimossa, la coda di messaggi (e ogni messaggio che non è ancora stato ricevuto) rimarrà nello spazio di memoria riservato dal kernel a questa mailbox. Per rimuovere la coda di messaggi, e cancellare i messaggi non letti che vi sono salvati, è necessario invocare `msgctl()`, come segue:

```
struct msgid_ds dummyParam;
status = msgctl(msqid, IPC_RMID, &dummyParam);
```

Il terzo parametro è necessario perché la funzione lo richiede, ma viene utilizzato solamente se il programmatore desidera richiamare alcune statistiche a proposito dell'uso della coda di messaggi. Ciò avviene impostando `IPC_STAT` come secondo parametro.

Tutti i programmi dovrebbero includere i tre file d'intestazione seguenti, che si trovano in `/usr/include/sys: ipc.h`, `types.h`, e `msg.h`. Va menzionato, a questo punto, un dettaglio di implementazione della coda di messaggi che potrebbe forse indurre in confusione. Dopo che una mailbox viene rimossa attraverso `msgctl()`, i tentativi di creare un'altra mailbox con lo stesso nome utilizzando `msgget()` genereranno solitamente uno `msqid` differente.

Parte 3: Creazione dei processi

Ciascun processo esterno, come pure il server centrale, creerà la propria mailbox con il nome $X + i$, dove i è un identificatore numerico. Per i processi esterni si avrà $i = 1,...,4$, mentre per il processo centrale $i = 0$. Così, se X fosse 70, il processo centrale riceverebbe messaggi nella mailbox 70, e manderebbe le sue risposte alle mailbox 71-74. Il processo esterno 2 riceverebbe nella mailbox 72 e invierebbe alla mailbox 70, e così via. In questo modo ogni processo esterno si collega a due mailbox, e il processo centrale si collega a cinque mailbox. Se ciascun processo specifica `IPC_CREAT` quando invoca `msgget()`, il primo processo che invoca `msgget()` crea effettivamente la mailbox, mentre le chiamate successive a `msgget()` creano un collegamento alla mailbox esistente. Il protocollo per la rimozione dovrebbe fare in modo che la mailbox/coda di messaggi dove ogni processo è in ascolto sia l'unica che il processo rimuove, attraverso `msgctl()`.

Ogni processo esterno sarà identificato univocamente tramite un parametro passato dalla riga di comando. Il primo parametro da passare a ogni processo esterno sarà la sua temperatura iniziale; il secondo parametro sarà il suo id numerico: 1, 2, 3 oppure 4. Per il server centrale basterà un unico parametro, cioè la sua temperatura iniziale. Assumendo che

il nome eseguibile del processo esterno sia `external` e quello del server centrale sia `central`, è possibile invocare tutti e cinque i processi come segue:

```
./external 100 1 &
./external 22 2 &
./external 50 3 &
./external 40 4 &
./central 60 &
```

Parte 4: Suggerimenti per l'implementazione

È meglio cominciare a implementare l'invio di un singolo messaggio dal processo centrale a un dato processo esterno, e viceversa, prima di provare a scrivere tutto il codice per risolvere questo problema. Sarebbe anche consigliabile controllare tutti i valori restituiti dalle quattro chiamate di sistema che gestiscono le code di messaggi, per verificare le richieste non andate a buon fine, e visualizzare un messaggio appropriato sullo schermo dopo che ciascuna chiamata è stata eseguita con successo. Il messaggio dovrebbe indicare che cosa è stato eseguito e da chi, ad esempio, "la mailbox 71 è stata creata dal processo esterno 1", "messaggio proveniente dal processo esterno 2 ricevuto dal processo centrale", e così via. Questi messaggi possono essere rimossi o commentati dopo la risoluzione del problema. I processi dovrebbero anche verificare di aver ricevuto il numero corretto di parametri dalla riga di comando (grazie al parametro `argc` in `main()`). Infine, eventuali messaggi estranei che risiedono nella coda possono far apparire erronei alcuni processi cooperanti che funzionano correttamente. Per questa ragione, conviene rimuovere tutte le mailbox che hanno rilevanza per il progetto, in modo da assicurare che le mailbox siano vuote prima dell'inzio del processo stesso. Il modo più semplice per fare ciò è utilizzare il comando `ipcs` per ottenere un elenco di tutte le code di messaggi e poi il comando `ipcrm` per rimuovere tutte le code di messaggi esistenti. Il comando `ipcs` elenca lo `msqid` di tutte le code di messaggi presenti sul sistema. Si può quindi utilizzare `ipcrm` per rimuovere le code di messaggi aventi un determinato `msqid`. Ad esempio, se nell'output di `ipcs` appare `msqid` 163845, la coda avente id 163845 può essere cancellata con il seguente comando:

```
ipcrm -q 163845
```

3.8 Note bibliografiche

La comunicazione tra processi nel sistema RC 4000 è trattata in [Binch-Hansen 1970]. [Schlichting e Schneider 1982] descrivono le primitive per lo scambio asincrono di messaggi. Il meccanismo IPC implementato a livello utente è descritto da [Bershad et al. 1990].

[Gray 1977] presenta i dettagli della comunicazione tra processi in UNIX. [Barrera 1991] e [Vahalia 1996] descrivono la comunicazione tra processi nel sistema Mach. [Solomon e Russinovich 2000] e [Stevens 1999] delineano la comunicazione tra processi in Windows 2000 e UNIX, rispettivamente. Hart [2005] esamina in dettaglio la programmazione nei sistemi Windows.

L'implementazione del meccanismo RPC è analizzata in [Birrell e Nelson 1984]. Il progetto di un meccanismo RPC affidabile è trattato da [Shrivastava e Panzieri 1982], mentre [Tay e Ananda 1990] offrono una panoramica di RPC. [Stankovic 1982] e [Staunstrup 1982] mettono a confronto la comunicazione basata su chiamate di procedura e quella fondata sullo scambio di messaggi. Harold [2005] analizza la programmazione delle socket in Java. Hart[2005] e Robbins e Robbins [2003] trattano il tema delle pipe, sia nei sistemi Windows sia nei sistemi UNIX.