Stefano Russo
Dipartimento di Informatica e Sistemistica
Università di Napoli "Federico II"

Meccanismi di gestione della concorrenza nelle applicazioni UNIX

Primitive UNIX per la gestione dei processi

- Per concorrenza si intende l'**esecuzione simultanea, reale o apparente, di più elaborazioni**

- Per sviluppare una applicazione concorrente, UNIX mette a disposizione le seguenti *system calls*:

- **fork** per creare un nuovo processo duplicando il processo chiamante

- **exec** per eseguire un programma memorizzato in un file eseguibile

- **wait** per sospendere l'esecuzione di un processo in attesa della terminazione di un processo figlio

- **exit** per terminare volontariamente un processo

La primitiva fork

- Tramite la *system call* UNIX fork un processo può creare un processo figlio che esegue concorrentemente con esso

- Il processo figlio è una copia del padre, sia per quanto riguarda il codice eseguito che per quanto riguarda l'ambiente (variabili, descrittori di file aperti, ecc.)

- I due processi sono indipendenti nel senso che ciascuno opera su proprie variabili

- Padre e figlio si distinguono perchè la fork ritorna "magicamente" due valori diversi:
  - zero nel processo figlio
  - l'identificativo UNIX del processo figlio (intero positivo) nel padre

```
main() {
  int pid; /* process identifier */
  pid = fork();
  if(pid < 0) {
/* situazione d'errore - impossibile generare il figlio */
  } else if (pid > 0) {
/* codice eseguito solo dal padre */
  } else {
/* codice eseguito solo dal figlio */
  }
}
```

- Il processo figlio ha le seguenti caratteristiche:
  - stesso codice del padre (codice condiviso)
  - area dati e stack con lo stesso contenuto al momento della fork, ma distinti

- stesso albiente d’esecuzione del padre (file aperti, direttorio corrente, ecc.)
- stesso contesto hardware d’esecuzione (PC, registri) al momento della fork

La primitiva exit

- La terminazione di un processo può essere

  involontaria azioni non consentite (es.: indirizzi scorretti)
  interruzione da tastiera da paarte dell'utente
  segnali da altri processi (primitiva kill)

  volontaria alla conclusione de codice
  eseguendo exit

- La *system call* exit è usata da un processo figlio per terminare in modo volontario l'esecuzione e ritornare un codice di terminazione al padre

```
int status;

exit(status);
```

- status è restituito (a cura del sistema operativo) al processo padre, se sta attendendo su una wait

- Per convenzione il valore zero indica una terminazione normale

La primitiva wait

- La *system call* wait è usata da un processo padre per attendere la terminazione di uno qualsiasi dei processi figli

```
int pid, status;

pid = wait(&status);
```

- Se sono state fatte più fork, non è possibile specificare l'attesa di un determinato figlio

- status contiene l'informazione circa la terminazione del figlio

- wait è **sospensiva** se ci sono processi figli in esecuzione, **non sospensiva** se tutti i figli sono terminati

- Se il figlio termina prima che il padre abbia eseguto wait, passa nello stato **ZOMBIE** (non sa risorse); viene rimosso dal sistema quando il padre esegue la wait

- Se il padre termina prima dei figli, questi (compresi gli zombie) vengono "adottati" dal processo INIT (inizializzazione di UNIX)

La primitiva exec

- La *system call* fork consente a padre e figlio di eseguire **solo lo stesso codice**

- Con la *system call* exec un processo può mandare in esecuzione un programma memorizzato in un file eseguibile

- L'effetto di exec è di rilasciare area codice ed area dati del processo chiamante, sostituendola con quella del nuovo programma

- Il resto del processo (pid, file aperti, ecc.) resta invariato; in altri termini, il programma eseguito eredita l'ambiente di chi invoca la exec

- Esistono diverse versioni di procedure di libreria che chiamano la exec, e che differiscono per il formato degli argomenti (execl, execv, execve)

execl(pathname, filename, arg1, ... , (char *)0);

- Non si ha ritorno al codice che ha invocato la exec, tranne in caso d'errore (es.: file non trovato, o non eseguibile)

I/O asincrono in UNIX

- Si può gestire la concorrenza nelle applicazioni, oltre che con più processi, gestendo più operazioni di ingresso/uscita simultaneamente
- UNIX BSD offre la primitiva di sistema select per gestire l'I/O concorrente
- Un processo che invoca select chiede al sistema operativo qual è il dispositivo (file o periferica) di I/O che diviene pronto per primo
- La select su un insieme di dispositivi ritorna appena uno di essi è pronto; il processo legge quindi da quel dispositivo
- Si consideri ad es. un programma cliente che deve leggere e visualizzare i caratteri ricevuti da una connessione TCP con un servente, e contemporaneamente deve prevedere che l'utente digiti un comando al terminale
- Esso non può sospendersi in attesa di un comando dell'utente, perchè deve continuare a visualizzare i caratteri ricevuti dal servente ...
- ma non può neppure sospendersi in una read sulla connessione, che è anch'essa bloccante
- Il problema è che il programma non sa quale input arriverà prima
- L'uso di select in questo caso consente di leggere dal primo dispositivo pronto tra terminale e connessione TCP

- In generale, la select consente di implementare serventi concorrenti monoprocesso, che gestiscono più connessioni con i clienti simultaneamente

Sommario

- Concorrenza vuol dire esecuzione simultanea, reale o apparente, di più elaborazioni
- In ambiente UNIX, i programmatori utilizzano le primitive di sistema *fork*, *wait*, *exit*, *exec*, *select* per gestire la concorrenza
- Un processo crea un altro processo copia di sé stesso tramite la primitiva *fork*
- Con la primitiva *wait*, un processo può sincronizzarsi con la terminazione di un processo generato in precedenza con la *fork*
- Un processo può eseguire il codice di un programma differente invocando la primitiva *exec*
- La primitiva UNIX *select* consente di gestire la concorrenza nelle operazioni di I/O su più dispositivi o canali di comunicazione