

Processi

- Concetto di processo
- Scheduling dei processi
- Operazioni sui processi
- Stati dei processi
- Esempio in Unix

1

Concorrenza

- Un sistema operativo consiste in un gran numero di attività che vengono eseguite più o meno contemporaneamente dal processore e dai dispositivi di un elaboratore
- Senza un modello adeguato la coesistenza delle diverse attività sarebbe difficile da descrivere e realizzare
- Il modello che è stato realizzato a questo scopo prende il nome di **modello concorrente** ed è basato sul concetto di **processo** (programma in esecuzione)
- In tale modello i processi vengono eseguiti in parallelo con parallelismo reale (multiprocessori) o apparente (multiprogrammazione su una singola CPU)
- La concorrenza studia i problemi legati all'esecuzione parallela di processi

2

Processi e programmi

- Definizione di processo
 - È un'attività controllata da un programma che si svolge su un processore (o più semplicemente un programma in esecuzione)
- Un processo non è un programma:
 - Un programma specifica la sequenza di esecuzione di un'insieme di istruzioni ma non la distribuzione nel tempo dell'esecuzione
 - Un processo rappresenta il modo in cui un programma viene eseguito nel tempo

3

Processi e programmi

- Più processi possono eseguire lo stesso programma (ad es. più utenti possono usare un browser per il web, un singolo utente può eseguire varie istanze dello stesso programma)
- Ogni istanza viene considerata come un processo separato anche se possono condividere lo stesso codice mentre (in generale) i dati, l'immagine e lo stato rimangono separati
- Il sistema operativo mantiene le informazioni sui singoli processi in strutture dati chiamate **Process Control Block (PCB)**
- Inoltre il sistema operativo tiene traccia di tutti i processi del sistema tramite la **Tabella dei Processi** (solitamente una tabella di puntatori a PCB).
- Ogni processo ha una entry nella tabella dei processi che punta al suo PCB.

4

Process Control Block (PCB)

Un processo può essere totalmente descritto dalle seguenti componenti

- Dati identificativi del processo e dell'utente e/o processo che l'ha generato
- Immagine di stato nel processore
 - Contenuto dei registri generali, del program counter e dei registri di stato
- Immagine di memoria
 - Segmento del codice da eseguire (lista di istruzioni caricate in memoria)
 - Segmento dati su cui operare
 - Stack di lavoro per la gestione delle chiamate di funzione, passaggio di parametri e variabili locali

5

- Informazioni di controllo del processo
 - Stato del processo (es. in esecuzione, pronto (ready), in attesa)
 - Informazioni per l'esecuzione in CPU (es. priorità)
- Informazioni per la gestione della memoria (es. valori dei registri base e limite dei segmenti utilizzati)
- Informazioni sull'utilizzo delle risorse (es. working directory, file aperti, device allocati)
- Informazioni di accounting (es. tempo di inizio dell'esecuzione e tempo cumulativo di CPU)
- Informazione per la comunicazione tra processi (stato dei segnali, semafori, etc.)

Scheduling di processi

- L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'uso della CPU
- L'obiettivo del time-sharing è quello di commutare l'uso della CPU tra diversi processi in modo da far interagire gli utenti con ciascun programma in esecuzione
- Lo **scheduler** gestisce l'avvicendamento dei processi in CPU
 - Decide quale processo deve essere in esecuzione ogni istante
 - Interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina
 - Interviene periodicamente per assicurare una corretta ripartizione nel tempo della CPU tra i vari processi

6

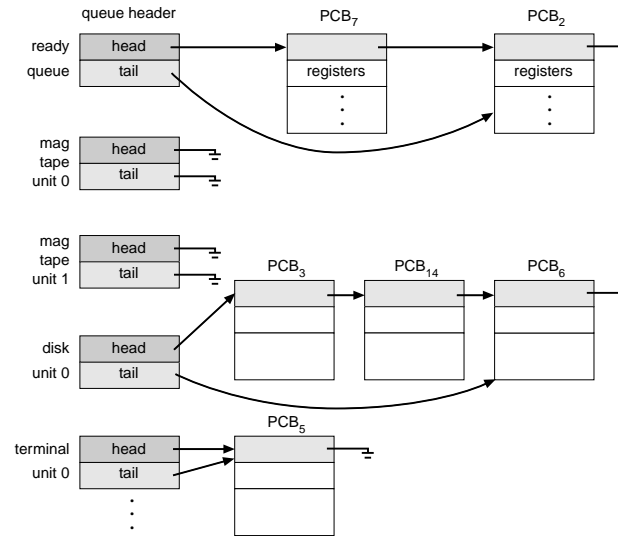
- Il sistema operativo utilizza un interrupt speciale chiamata interrupt di clock per risvegliare lo scheduler ogni tot di tempo (time slice/quanto di tempo)
- Terminologia (utilizzata anche per l'organizzazione di risorse aziendali, umane, ecc.)
 - Schedule: sequenza temporale di assegnazione della CPU ai processi
 - Scheduling: l'azione di calcolare uno schedule
 - Scheduler: software che calcola uno schedule

Code di scheduling dei processi

Il sistema operativo gestisce i processi tramite una serie di code di attesa per l'accesso alle risorse dell'elaboratore

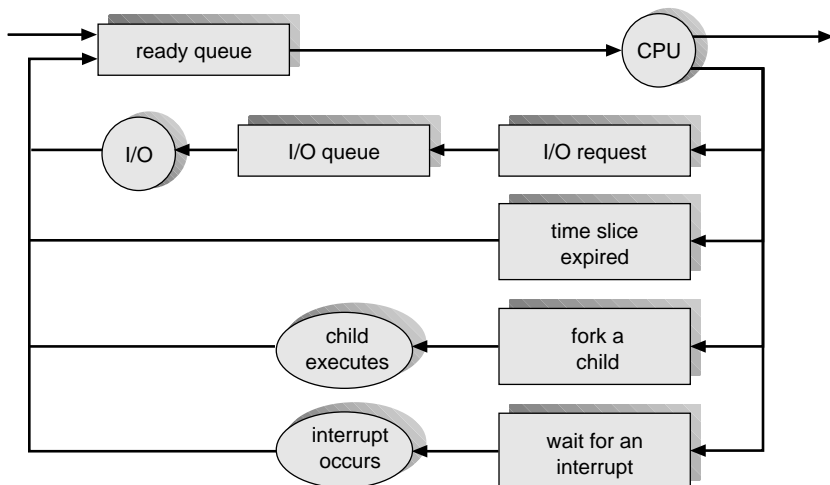
- La coda dei processi contiene (puntatori a) tutti i processi (cioè ai loro PCB) nel sistema e determina l'ordine di ingresso nel sistema
- La ready queue contiene (puntatori a) i processi residenti in memoria principale, pronti e in attesa di essere messi in esecuzione
- Le code dei dispositivi contengono (puntatori a) i processi in attesa di un dispositivo di I/O.

7



Migrazione dei processi tra le code

I processi, durante l'esecuzione, migrano da una coda all'altra

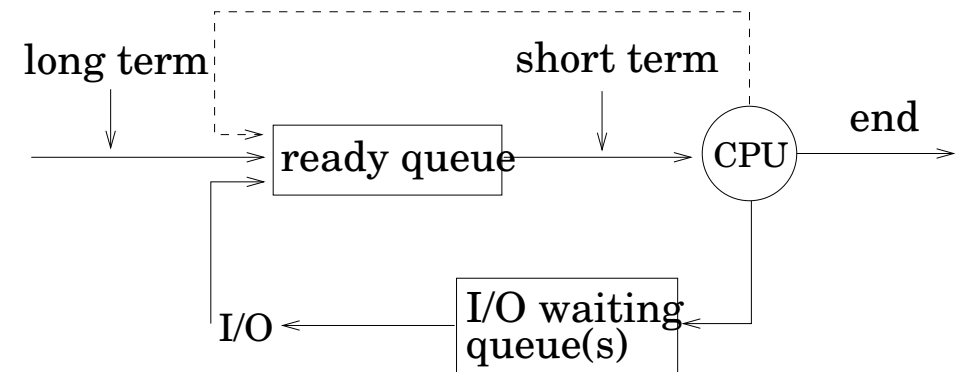


Gli scheduler scelgono quali processi passano da una coda all'altra.

8

Gli Scheduler

- Lo scheduler di lungo termine (o job scheduler) seleziona i processi da portare nella ready queue.
- Lo scheduler di breve termine (o CPU scheduler) seleziona quali processi ready devono essere eseguiti, e quindi assegna la CPU.



9

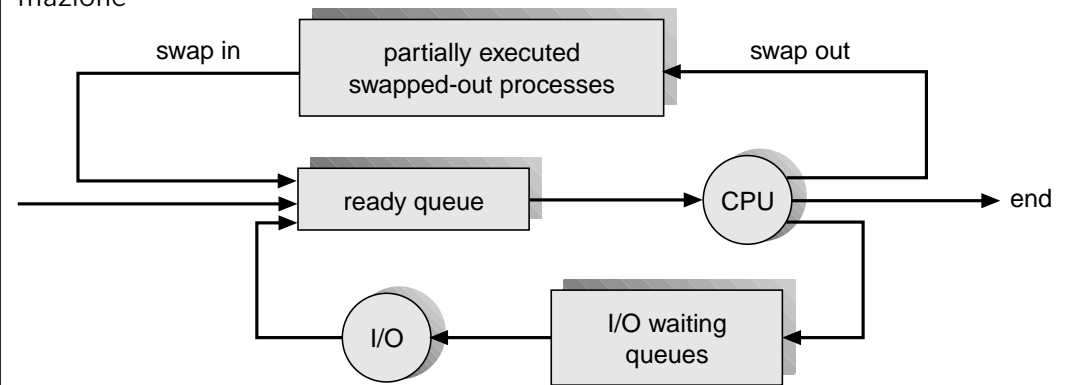
Gli Scheduler (Cont.)

- Lo scheduler di breve termine è invocato molto frequentemente (decine di volte al secondo) ⇒ deve essere veloce
- Lo scheduler di lungo termine è invocato raramente (secondi, minuti) ⇒ può essere lento e sofisticato
- I processi possono essere descritti come
 - I/O-bound: lunghi periodi di I/O, brevi periodi di calcolo.
 - CPU-bound: lunghi periodi di intensiva computazione, pochi (possibilmente lunghi) cicli di I/O.
- Lo scheduler di lungo termine controlla il grado di multiprogrammazione e il *job mix*: un giusto equilibrio tra processi I/O e CPU bound.

10

Gli Schedulers (Cont.)

Alcuni sistemi hanno anche lo *scheduler di medio termine* (o *swap scheduler*) sospende temporaneamente i processi per abbassare il livello di multiprogrammazione



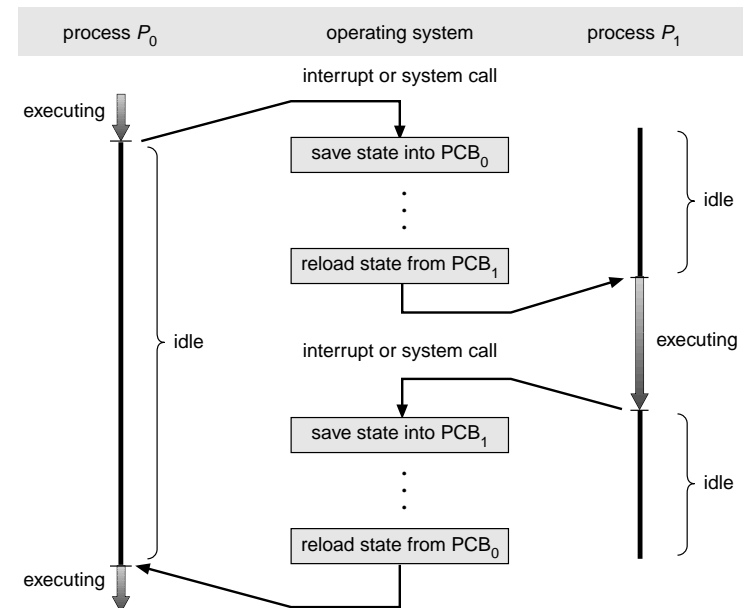
11

Fase di Context-Switch

- Quando la CPU passa ad un altro processo, il sistema deve salvare lo stato del vecchio processo (contesto di esecuzione) e caricare quello del nuovo processo
- Questa fase viene chiamata context-switch (cambiamento/switch del contesto)
- Lo stato del processo viene salvato nel PCB
- Il tempo di context-switch porta un certo overhead; il sistema non fa un lavoro utile mentre passa di contesto
- Può essere un collo di bottiglia per sistemi operativi ad alto parallelismo
- Il tempo impiegato per lo switch dipende dal supporto hardware (per il salvataggio dei valori dei registri, ecc.)

12

Context-Switch



13

Operazioni e ciclo di vita di un processo

- Creazione e terminazione di un processo
- Gerarchie di processi
- Cambiamento dello stato di un processo
- Comunicazione tra processi

14

Creazione dei processi

- Quando viene creato un processo?
 - Al boot del sistema (es. demoni di stampa, di rete, ecc.)
 - Su richiesta da parte dell'utente (es. esecuzione di un'applicazione)
 - Come risposta ad una richiesta di servizio ad un demone
 - In generale su esecuzione di una system call apposita
 - Ad esempio in Unix occorre sempre passare attraverso la system call *fork*

15

Gerarchie di processi

- La fase di creazione induce una naturale gerarchia, chiamata l'albero dei processi, basata sulla relazione padre (creatore) e figlio (nuovo processo) tra i processi gestiti dal sistema operativo
- Vi sono diverse alternative per gestire l'esecuzione dei processi in tale gerarchia:
 - Padre e figli sono in esecuzione concorrente
 - Il padre attende che i figli terminino per riprendere l'esecuzione
- diverse alternative per gestire le loro risorse (ad es. file aperti)
 - Padre e figli condividono le stesse risorse
 - I figli condividono un sottoinsieme delle risorse del padre
 - Padre e figli non condividono nessuna risorsa

16

- e diverse alternative per gestire il loro spazio di indirizzamento
 - I figli duplicano lo spazio di indirizzi del padre
 - I figli caricano sempre un programma subito dopo la creazione

Terminazione dei Processi

- Terminazione volontaria—normale o con errore. I dati di output vengono ricevuti dal processo padre
- Terminazione involontaria: errore fatale (superamento limiti, operazioni illegali, ...)
- Terminazione da parte di un altro processo (uccisione)
- Terminazione da parte del kernel (es.: il padre termina, e quindi vengono terminati tutti i discendenti: terminazione *a cascata*)
- In ogni caso le risorse del processo sono deallocate dal sistema operativo.

17

Stato del processo

Durante l'esecuzione, un processo cambia *stato*.

In *generale* si possono individuare i seguenti stati:

new: il processo è appena creato

running: istruzioni del programma vengono eseguite da una CPU.

waiting: il processo attende qualche evento

ready: il processo attende di essere assegnato ad un processore

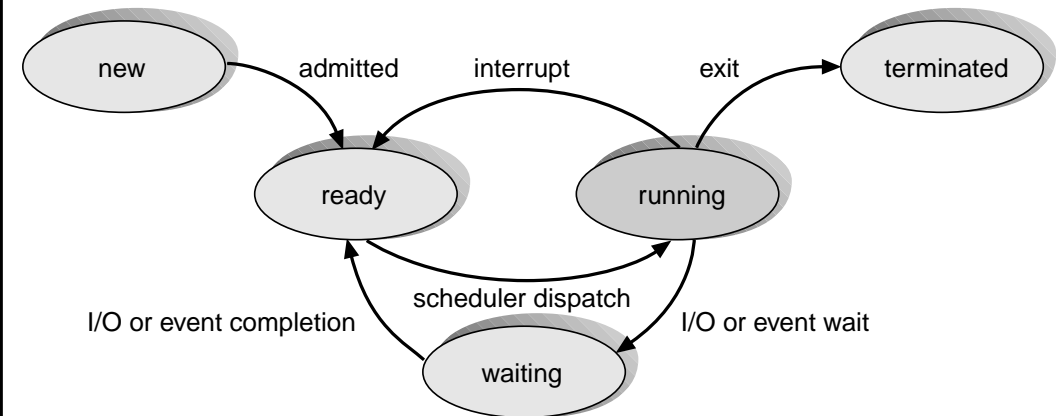
terminated: il processo ha completato la sua esecuzione

18

In alcuni sistemi si parla anche di stato **zombie**: il processo ha terminato la sua esecuzione ma il sistema operativo mantiene ancora attiva la sua entry nella tabella dei processi con informazioni che possono essere utili ad esempio al padre.

Il passaggio da uno stato all'altro avviene, ad esempio, in seguito a interrupt, richieste di risorse non disponibili, selezione da parte dello scheduler, ...

Diagramma degli stati



19

Cooperazione tra processi

- Un processo in esecuzione si dice cooperante quando può influenzare o può essere influenzato dall'esecuzione di un altro processo
- La gestione di processi cooperanti è necessaria quando si devono condividere risorse
- può essere inoltre utile
 - per permettere forme di comunicazione tra processi, ad es., tramite memoria condivisa,
 - per accelerare il calcolo (suddivisione di un task in sottotask paralleli opportunamente coordinati tra loro)
- Il tipico esempio di processi cooperanti è legato al problema del produttore e del consumatore

20

• Problema del Produttore-Consumatore

- Il produttore (ad es. editor) produce informazioni (ad es. caratteri) e le memorizza in un buffer di capacità limitata che sono consumate dal consumatore (es. stampante).
- In principio tali operazioni possono essere eseguite da processi eseguiti in parallelo. Abbiamo bisogno però di sincronizzare i processi (se l'editor non produce caratteri il consumatore deve attendere l'input)
- Se il buffer è pieno il produttore deve aspettare il consumatore prima di inserire nuovi item (altrimenti rischia di sovrascrivere item non consumati)
- In un sistema con memoria condivisa i due processi potrebbero utilizzare una variabile comune (indice che indica quanti elementi ci sono nel buffer) per sincronizzare le operazioni
- Vedremo in seguito i problemi legati alla sincronizzazione di processi cooperanti

Comunicazione tra processi

- I sistemi operativi solitamente forniscono diversi meccanismi di comunicazione tra processi indipendenti (senza memoria condivisa) che permettono di coordinare azioni eseguite in parallelo
- Comunicazione diretta:
 - un processo che intenda comunicare deve conoscere il nome del ricevente;
 - il ricevente può indicare il nome del trasmittente o attendere un messaggio da un processo qualsiasi
 - la comunicazione avviene scambio di messaggi su un canale stabilito automaticamente tra coppie di processi

21

• Comunicazione indiretta:

- si inviano i messaggi a delle porte (mailbox) dalle quali i processi possono successivamente prelevarli
- si stabilisce un canale tra due processi se essi conoscono la stessa porta
- un canale può essere associato a più processi
- Lo scambio di messaggi può essere
 - sincrono (bloccante) per il trasmittente: aspetta che il messaggio venga ricevuto
 - sincrono (bloccante) per il ricevente: aspetta di ricevere un messaggio non nullo prima di proseguire l'esecuzione
 - asincrono (non bloccante) per il trasmittente: prosegue immediatamente l'esecuzione dopo l'invio
 - asincrono (non bloccante) per il ricevente: riceve un messaggio valido o nullo e prosegue immediatamente l'esecuzione

- Solitamente si utilizzano delle code per non perdere i messaggi inviati ma non ancora processati dal ricevente
 - capacità zero: il canale non può avere messaggi pendenti
 - capacità limitata: il canale può avere un numero finito di messaggi pendenti (il trasmittente potrebbe bloccarsi se la coda è piena)
 - capacità potenzialmente illimitata: il canale può avere un numero arbitrario di messaggi pendenti (il trasmittente non si blocca mai)

Meccanismi di comunicazione Client-server

- Socket: una socket è un'estremità di un canale di comunicazione (es. identificata da indirizzo IP + numero porta). Il server rimane in attesa delle richieste dei clienti su una porta specificata; quando riceve una richiesta, se accetta la connessione proveniente dal cliente, si stabilisce la comunicazione attraverso una socket che il server usa per inviare/ricevere dati dal cliente
- RPC (Remote Procedure Call): si invoca una procedura su un sistema remoto. Un demone RPC in ascolto su una porta speciale del server aspetta messaggi (nome procedura da eseguire e parametri), invoca la procedura e restituisce il risultato

22

Processi in UNIX

- Creazione dei processi: fork e execve
- Strutture dati per gestire processi
- Ciclo di vita
- Context switch e gestione interruzioni

23

Processi in UNIX tradizionale

- Ogni processo UNIX è identificato dal PID - process identifier
- Ogni processo ha un spazio di indirizzamento separato e quindi non può vedere le zone di memoria dedicate agli altri processi.
- I processi possono essere eseguiti in due modalità: modalità user e kernel
- Un processo cambia modalità quando occorre un'interrupt o quando invoca una system call (che genera una trap)
- Per gestire interrupt e trap il sistema operativo opera nello stesso contesto del processo utente cioè salva il contesto del processo utente e lo sostituisce con un contesto relativo al programma di gestione della trap/interrupt)
- Si utilizzano inoltre due stack separati per le chiamate di funzione: user stack e kernel stack
- Il kernel stack viene utilizzato per gestire chiamate di funzione durante l'esecuzione in modalità kernel

24

Componenti di un processo UNIX

- Un processo UNIX ha tre segmenti:
 - Stack di attivazione delle procedure (cambia dinamicamente)
 - Dati (cambia dinamicamente)
 - * Dati inizializzati e non inizializzati al caricamento del programma
 - * Heap per la gestione delle strutture dati dinamiche
 - Text: codice eseguibile. Non modificabile, protetto in scrittura.

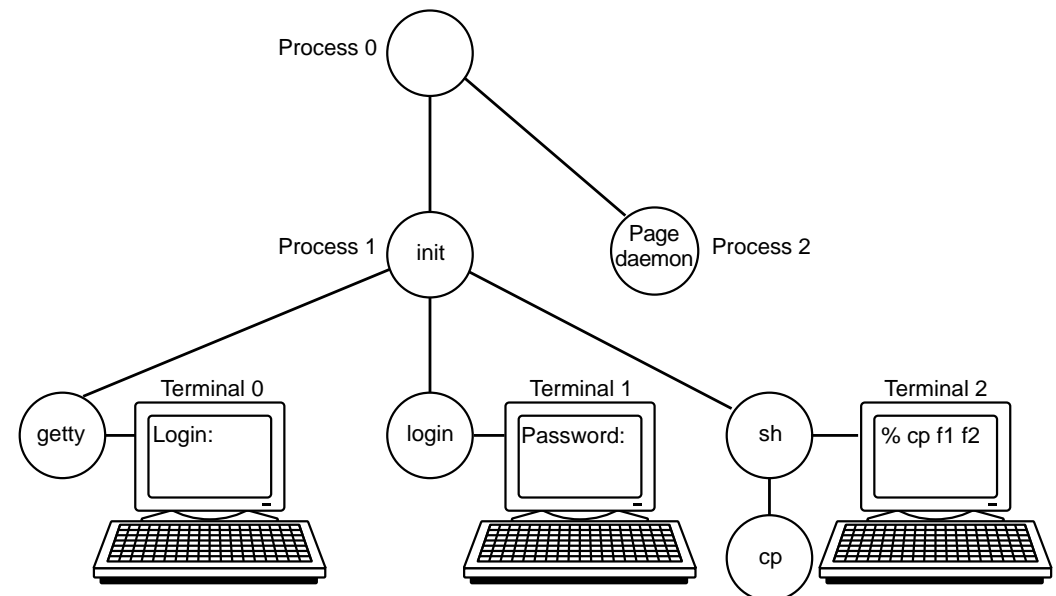
25

- L'indirizzamento della memoria è virtuale (i processi pensano di avere a disposizione tutta la memoria)
 - il codice, i dati e lo heap sono memorizzati nella parte iniziale della memoria virtuale (da indirizzo 0 in su),
 - lo stack nella parte finale (dall'indirizzo LAST in giù)
- Il mapping tra indirizzi virtuali e fisici viene gestito dal sistema operativo
- Il gap tra heap e stack viene utilizzato per allocare nuova memoria dinamicamente (ad es. quando si crea una nuova cella in una lista con puntatori attraverso la system call *malloc*)

Alcuni Processi Unix

- Ogni processo Unix, tranne il processo con PID=0, viene creato attraverso la system call *fork*
- Il processo 0 viene creato è un processo speciale creato dal boot di sistema
 - dopo aver creato (tramite *fork*) il processo *init* (PID=1) il processo 0 diviene il processo *swapper* che gestisce lo spostamento di processi da memoria a disco che a sua volta utilizza il processo di gestione della paginazione (demone delle pagine) - vedremo meglio questi concetti più avanti -
- Il processo *init* è l'antenato di tutti gli altri processi
- Inizialmente genera i processi che servono al funzionamento del sistema operativo, ad es., spawning dei demoni e dei processi per l'interazione con l'utente *getty*, *login*, *shell*
- Inoltre adotta tutti i figli che diventano *orfani* (termina il padre) (in questo modo non si perde mai la gerarchia ad albero dei processi)

26



Operazioni per creazione di processi

- L'unico modo di creare nuovi processi in Unix è attraverso la funzione di sistema *fork*
- La chiamata *fork()* dal processo *P* (padre) crea un nuovo processo *F* (figlio) che viene eseguito in parallelo con il padre
- Dopo la creazione del figlio, padre e figlio condividono lo stesso codice, inoltre il figlio ha una copia dei dati e del program counter del padre
- Il figlio proseguirà l'esecuzione a partire dall'istruzione che segue la *fork* (cioè lo stesso punto nel quale si trova il padre)

27

- Per distinguere padre e figlio se la chiamata *fork()* ha successo allora restituisce:
 - il PID del figlio al processo padre
 - 0 al figlio
- Il valore di ritorno di *fork* viene usato quindi per ridefinire il comportamento del figlio
- Tipicamente infatti il figlio sostituirà l'immagine del processo padre con un nuovo programma attraverso la chiamata di sistema alla funzione *execve* (che prende come parametro il nome di un file eseguibile)
- La funzione *execve()* sostituisce dati, codice e stack con quelli del nuovo programma da eseguire nel contesto del processo figlio
- Il padre può aspettare la terminazione del figlio utilizzando la funzione di sistema *waitpid*

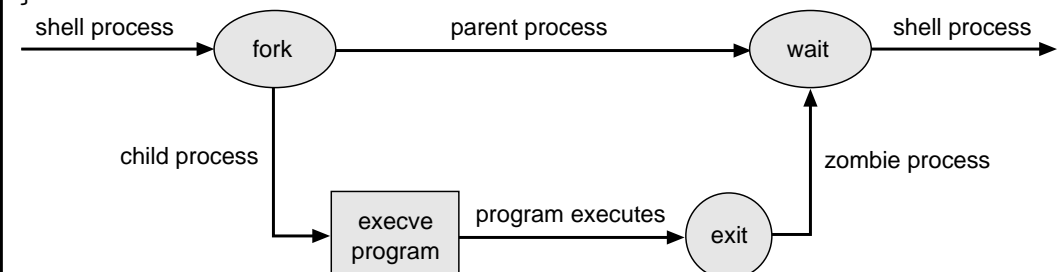
Schema di chiamata fork da programma

```
pid = fork();
if (pid < 0) {
    /* fork fallito */
} else if (pid > 0) {
    /* codice eseguito solo dal padre */
} else {
    /* codice eseguito solo dal figlio */
}
/* codice eseguito da entrambi */
```

28

Esempio: ciclo fork/wait di una shell

```
while (1) {
    read_command(commands, parameters);
    if (fork() != 0) { /* parent code */
        waitpid(-1, &status, 0);
    } else { /* child code */
        execve(command, parameters, NULL);
    }
}
```



29

Gestione e implementazione dei processi in UNIX

- In UNIX, l'utente può creare e manipolare direttamente più processi
- I processi sono rappresentati da *process control block*
 - Il PCB di ogni processo è memorizzato in parte nel kernel (*process structure, text structure*), in parte nello spazio di memoria del processo (*user structure*)
 - L'informazione in questi blocchi di controllo è usata dal kernel per il controllo dei processi e per lo scheduling.
- Inoltre si utilizzano i *segnali* (gestiti dal kernel attraverso opportune system call) per avvisare i processi del verificarsi di eventi asincroni quali ad esempio
 - segnale di terminazione di un processo inviato al padre (es. system call *exit, signal*)

30

- segnale riguardanti eccezioni e condizioni di errore indotte da un processo/chiamata di sistema
- segnali di risveglio inviati da processi utente
- terminazione di un processo tramite system call *kill*
- ...
- Il kernel tiene traccia dei segnali inviati ai processi utilizzando opportuni flag della tabella dei processi (tale gestione avviene durante l'esecuzione in modo kernel)
- I segnali vengono interpretati solo quando un processo torna da modo kernel a modo utente (e non quando si è in modo kernel)

Strutture dati per i processi Unix

- La struttura base più importante è la *process structure*: contiene
 - stato del processo
 - puntatori alla memoria (segmenti, *u-structure*, *text structure*)
 - identificatori del processo
 - identificatori dell'utente (per la gestione dei segnali)
 - informazioni di scheduling (e.g., priorità)
 - segnali non gestiti
- La *text structure* (struttura del codice)
 - è sempre residente in memoria
 - memorizza quanti processi stanno usando il segmento codice (permette quindi condivisioni del codice)
 - contiene dati relativi alla gestione della memoria virtuale per il codice

31

Process Control Block (Cont.)

- Le informazioni sul processo che sono richieste solo quando il processo è residente sono mantenute nella *user structure* (o *u-structure*). Fa parte dello spazio indirizzi modo user, read-only (ma scrivibile dal kernel) e contiene (tra l'altro)
 - identificatore utente e gruppo
 - risultati/errori delle system call
 - tabella dei file aperti
 - limiti del processo

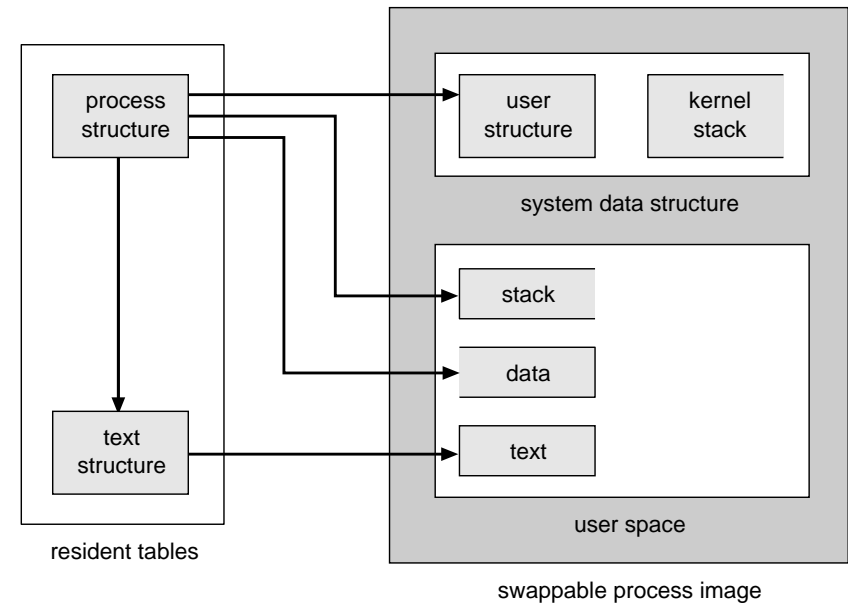
32

Segmenti dei dati di sistema

- La maggior parte della computazione viene eseguita in user mode; le system call vengono eseguite in kernel mode
- Le due fasi di un processo non si sovrappongono mai: un processo si trova sempre in una o l'altra fase
- Per l'esecuzione in modo kernel, il processo usa uno stack separato (*kernel stack*), invece di quello del modo utente
- Kernel stack + u-structure = *system data segment* del processo

33

Parti e strutture di un processo



34

Creazione di un processo

- La **fork** alloca una nuova process structure per il processo figlio
 - nuove tabelle per la gestione della memoria virtuale
 - nuova memoria viene allocata per i segmenti dati e stack
 - i segmenti dati e stack e la u-structure vengono copiati da quelli del padre, in questo modo vengono preservati i file aperti, UID e GID, ecc.
 - il codice viene condiviso, puntando alla stessa text structure
- La **execve** non crea nessun nuovo processo: semplicemente, i segmenti dati, codice e stack vengono rimpiazzati con quelli del nuovo programma

35

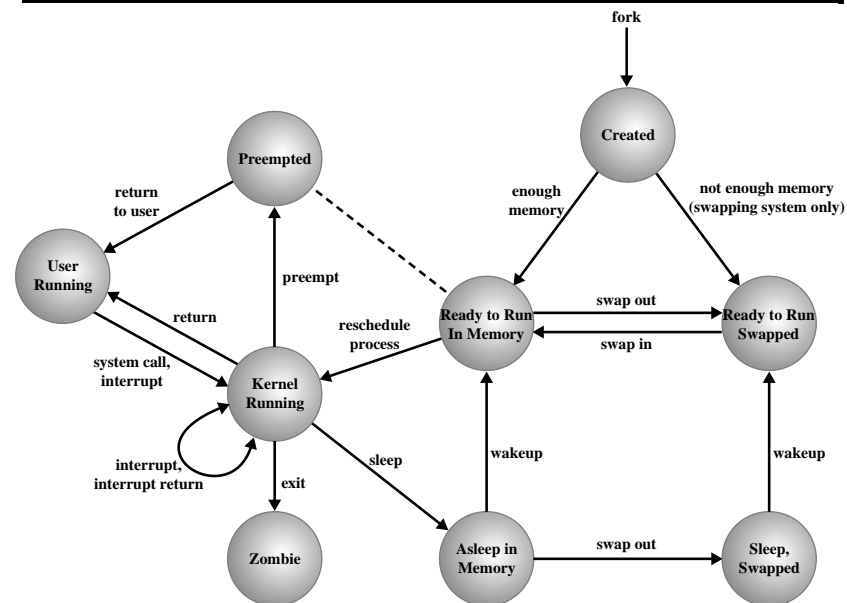
Ciclo di vita di un processo Unix

- Il processo entra nel sistema nello stato *Created* quando il padre crea il processo tramite la chiamata a `fork`
- Il processo muove in uno stato in cui è pronto a partire, ad es., *Ready to run in memory*
- Se viene selezionato dallo scheduler muove nello stato *Kernel running* (esecuzione in Kernel mode) dove termina la sua parte di *fork*.
- Quando termina l'esecuzione della chiamata di sistema può passare allo stato *User running* (esecuzione in modo User) oppure potrebbe passare nello stato *Preempted* (cioè lo scheduler ha selezionato un'altro processo)
- Dallo stato *User running* può passare allo stato *Kernel running* a seguito di un'altra chiamata di sistema oppure di un'interrupt, ad es., di clock

36

- Dopo aver eseguito la routine di gestione dell'interrupt di clock il Kernel potrebbe schedare un altro processo da mandare in esecuzione e quindi il processo in questione passerebbe allo stato *Preempted*
- Lo stato *Preempted* enfatizza il fatto che i processi Unix possono essere prelatzionati solo quando tornano da modo Kernel a modo utente
- Se durante l'esecuzione in modo Kernel (ad es. di una system call) il processo deve eseguire operazioni di I/O passa allo stato *Asleep in memory* per essere risvegliato successivamente e passare nello stato *Ready to run*
- Gli stati con etichetta *Swapped* corrispondono a situazioni nelle quali il processo non è più fisicamente in memoria centrale (ad es. non c'è abbastanza memoria per gestire i processi in multitasking)

Diagramma degli stati di un processo in UNIX



37

User and Kernel Mode

- I processi Unix possono operare in modo user e kernel: cioè il kernel esegue nel contesto di un processo le operazioni per gestire chiamate di sistema e interrupt
 - Alla partenza del sistema il codice del kernel viene caricato in memoria principale (con strutture dati (tabelle) necessarie per mappare indirizzi virtuali kernel in indirizzi fisici)
 - Un processo in esecuzione in modo user non può accedere allo spazio di indirizzi del kernel
 - Quando un processo passa ad eseguire in modo kernel tale vincolo viene rilasciato: in questo modo si può eseguire codice del kernel (routine di gestione di interrupt/codice di una chiamata di sistema) nel contesto del processo utente
- Il contesto di un processo: contesto utente (codice, dati, stack), registri, e contesto kernel (entry nella tabella dei processi, u-area, stack kernel)

38

Livelli di contesto

- La parte dinamica del contesto di un processo (kernel stack, registri salvati) è organizzata a sua volta come stack con un numero di posizioni che dipende dai livelli di interrupt diversi ammessi nel sistema
- Ad esempio se il sistema gestisce interrupt software, interrupt di terminali, di dischi, di tutte le altre periferiche, e di clock: avremo al più sette livelli di contesto
 - Livello 0: User
 - Livello 1: Chiamate di sistema
 - Livelli 2-6: Interrupt (l'ordine dipende dalla priorità associata alle interrupt)

39

Esempio di esecuzione nel contesto di un processo

- Il processo esegue una chiamata di sistema: il kernel salva il suo contesto (registri, program e stack pointer) nel livello 0 e crea il contesto di livello 1
- La CPU riceve e processa un interrupt di disco (il controllo viene fatto prima dell'esecuzione della prossima istruzione): il kernel salva il contesto di livello 1 (registri, stack kernel) e crea il livello 2 nel quale si esegue la routine di gestione dell'interrupt di disco
- La CPU riceve un interrupt di clock: il kernel salva il contesto di livello 2 (registri, stack kernel per la routine di gestione dell'interrupt disco) e crea il livello 3 nel quale si esegue la routine di gestione dell'interrupt di clock
- La routine termina l'esecuzione: il kernel recupera il livello di contesto 2 e così via
- Tutti questi passi vengono fatti sempre all'*interno dello stesso processo*: cambia solo la sua parte di contesto dinamica

40

Algoritmo di gestione delle interruzioni

- L'algoritmo del kernel per la gestione di un'interrupt consiste dei seguenti passi:
 - salva il contesto del processo corrente
 - determina fonte dell'interrupt (trap/interrupt I/O/ ecc)
 - recupera l'indirizzo di partenza della routine di gestione delle interrupt (dal vettore delle interrupt)
 - invoca la routine di gestione dell'interrupt
 - recupera il livello di contesto precedente
- Per motivi di efficienza parte della gestione di interruzioni e trap viene eseguita direttamente dalla CPU (dopo aver seguito un'istruzione): il kernel dipende quindi dal processore

41

Trap/Interrupt vs Context switch

- Il modo user/kernel permette al kernel di lavorare nel contesto di un altro processo senza dover creare nuovi processi kernel
- Con questo meccanismo un processo in modo kernel può svolgere funzioni logicamente collegate ad altri processi (ad es. la gestione dei dati restituiti da un lettore di disco) e non necessariamente collegate al processo che *ospita* momentaneamente il kernel
- Nota: La gestione di trap/interrupt si basa su una sorta di context switch all'interno di un processo: il controllo non passa ad un'altro processo ma è necessario salvare la parte corrente del contesto dinamico del processo all'interno dello stesso processo

42

Ma allora quando avviene un context switch tra processi?

- Il kernel vieta context switch arbitrari per mantenere la consistenza delle sue strutture dati
- Il controllo può passare da un processo all'altro in quattro possibili scenari:
 - Quando un processo si sospende
 - Quando termina
 - Quando torna a modo user da una chiamata di sistema ma non è più il processo a più alta priorità
 - Quando torna a modo user dopo che il kernel ha terminato la gestione di un'interrupt a non è più il processo a più alta priorità
- In tutti questi casi il kernel lascia la decisione di quale processo da eseguire allo scheduler

43