Appunti di Sistemi Operativi I

Colacel Alexandru Andrei

Disclaimer

Le fonti sono le slides del Prof. Tolomei e integrazioni con il libro *I moderni sistemi operativi* di Andrew S. Tanenbaum. Se diversamente verrà indicato a pié di pagina.

Indice

1 Introduzione **2**
- 1.1 Kernel/User mode e protezione della memoria . . . 2
- 1.2 Che cos'è un Sistema Operativo . . . 2
  - 1.2.1 L'OS come macchina estesa . . . 2
  - 1.2.2 L'OS come gestore delle risorse . . . 2
- 1.3 Analisi dell'hardware . . . 3
  - 1.3.1 System Bus . . . 3
  - 1.3.2 Dispostivi di I/O . . . 3
  - 1.3.3 Processori . . . 4
  - 1.3.4 Protezione delle system call . . . 5
  - 1.3.5 Passaggio di parametri . . . 6
  - 1.3.6 Timer, Istruzioni atomiche, Virtual Memory . . . 6
- 1.4 Progettazione e implementazione del sistema operativo . . . 7

2 Gestione dei Processi e thread **8**
- 2.1 Stato di esecuzione del processo . . . 9
- 2.2 Process Control Block (PCB) . . . 10
- 2.3 Creazione di processi . . . 11
  - 2.3.1 Risorse del padre vs figlio . . . 12
  - 2.3.2 Esecuzione del padre vs figlio . . . 12
- 2.4 Terminazione di processi . . . 15
- 2.5 Scheduling dei processi . . . 15
  - 2.5.1 Il Context Switch . . . 16
- 2.6 Communicazione dei processi . . . 16
- 2.7 Scheduling dei processi . . . 17
- 2.8 Scheduling Preemptive e Non-Preemptive . . . 18
  - 2.8.1 Problemi . . . 18
- 2.9 Il Dispatcher . . . 18
- 2.10 Definizioni utili . . . 18

3 Sincronizzazione tra Processi/Thread **20**

4 Gestione della Memoria **21**

5 Gestione dei Sistemi di I/O **22**

6 File System **23**

7 Advanced Topics **24**

1 Introduzione

Il programma con cui si interagisce può essere formato testo (quindi la **$SHELL**) oppure modalità **GUI** (Graphical User Interface) quando ci sono le icone. La maggior parte dei computer ha due modalità operative: kernel e utente. Il sistema operativo è il componente software di maggior importanza e viene eseguito in modalità kernel (detta anche Supervisor). In questo modo ha accesso a tutto l'hardware e può eseguire qualunque istruzione che la macchina sia in grado di svolgere. La modalità utente ha a sua disposizione solo un sottoinsieme di istruzioni.
Esiste una differenza tra il sistema operativo (che troverete scritto anche OS) e il normale software in modalità utente. Infatti l'utente non è livero di scrivere E.g un gestore degli interrupt ecc... perchè esistono protezioni hardware dell'OS stesso
Questa differenza è meno evidente nei sistemi embedded o sistemi integrati (che possono non avere la modalità kernel) o sistemi interpretati che usano interpreti e non hardware per separare i componenti

1.1 Kernel/User mode e protezione della memoria

Alcune istruzioni eseguite dalla CPU risultano essere più sensibili di altre. Affinché tali istruzioni privilegiate vengano utilizzate esclusivamente dal sistema operativo, la CPU può essere impostata in due modalità specifiche a seconda del programma in esecuzione. La CPU può essere quindi impostata in:

- Kernel mode ossia in modalità senza alcuna restrizione, permettendo l'esecuzione di qualsiasi istruzione (utilizzata dal sistema operativo)
- User mode, dove non sono possibili:
  - Accedere agli indirizzi riservati ai dispositivi di I/O
  - Manipolare il contenuto della memoria principale
  - Arrestare il sistema
  - Passare alla Kernel Mode
  - ...

Per poter impostare una delle due modalità, viene utilizzato un bit speciale salvato in un registro protetto: se impostato su 0 la CPU sarà in Kernel mode, mentre se impostato su 1 la CPU sarà in User mode

1.2 Che cos'è un Sistema Operativo

Il Sistema Operativo esegue fondamentalmente due funzioni non correlate. Da una parte fornisce ai programmatori funzioni di applicazioni un insieme di risorse astratte e dall'altra le risorse hardware.

1.2.1 L'OS come macchina estesa

L'archiettura è l'insieme delle istruzioni, organizzazione della memoria, I/0, e struttura dei bus. Per la gestione dell'hardware si utilizza un software chiamato **driver** che fornise l'interfaccia per la lettura e la scrittura senza che il programmatore si occupi dei dettagli. L'astrazione è la chiave per risolvere la complessità, una buona astrazione suddivide un'attività complessa in due attività più gestibili. La prima riguarda la definizione e l'implementazione delle astrazioni. La seconda riguarda l'impiego di queste astrazioni per risolvere problemi reali.

1.2.2 L'OS come gestore delle risorse

La gestione delle risorse include il multiplexing (condivisione) delle risorse in due modalità diverse: nel tempo e nello spazio.Quando una risorsa è condivisa temporalmente programmi o utenti diversi fanno a turno ad usarla in un certo arco di tempo finito. L'altro tipo di multiplexing è nello spazio. Pressuponendo che vi sia abbastanza memoria per gestire parecchi programmi in memoria contemporaneamente, specialmente se necessita solo di una piccola frazione del totale. Tutto questo solleva però problemi di equità, protezione ecc... risolvibili dall'OS.

1.3 Analisi dell'hardware

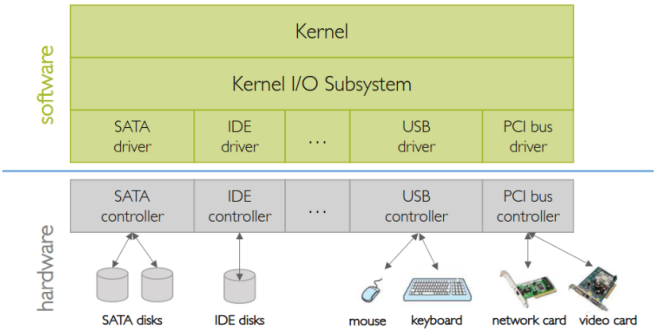

Figura 1: Alcuni dei componenti di un semplice personal computer

1.3.1 System Bus

Inizialmente veniva usato un unico bus per gestire tutto il traffico. Combina le funzioni di:

- Data bus effettivo di informazioni
- Address bus per determiare dove tali informazioni devono essere inviate
- Control bus per indicare quali operazioni dovrebbero essere eseguite

Sono stati aggiunti più bus dedicati per gestire il traffico da CPU a memoria e I/O, PCI, SATA, USB, ecc.

1.3.2 Dispostivi di I/O

Ogni dispositivo I/O è composto da due parti fondamentali il dispositivo fisico stesso e il controller del dispositivo (chip o set di chip che controllano una famiglia di dispositivi fisici).
Il sistema operativo comunica con un controller del dispositivo utilizzando un driver di dispositivo specifico.
La communicazione con i "Dispositivi di Controllo" ciacun controller del dispositivo ha un numero di registri dedicati per comunicare con esso:

- **Registri di stato**: forniscono alla CPU informazioni sullo stato del dispositivo I/O (ad esempio, inattivo, pronto per l'input, occupato, errore, transazione completata)
- **Registri di configurazione/controllo**: utilizzati dalla CPU per configurare e controllare il dispositivo
- **Registri dei dati**: utilizzati per leggere o inviare dati al dispositivo I/O

La CPU riesce ad indirizzare mettendo l'address di un byte di memoria nel address bus, specficando il segnale `READ` sul controllo del bus. Alla fine, la RAM risponderà con il contenuto della memoria sul Data bus.
La CPU può comunicare con un controller del dispositivo in due modi:

- I/O mappati alle porte che fanno riferimento ai registri del controller utilizzando un I/O separato spazio degli indirizzi.
  Il registro di ciascun controller del dispositivo I/O è mappato su una porta specifica (indirizzo).
  Richiede classi speciali di istruzioni CPU (ad es. IN/OUT). L'istruzione `IN` legge da un dispositivo I/O, mentre `OUT` scrive. Quando si usano le istruzioni IN o OUT, l' `M/#IO` non viene asserito, quindi la memoria non risponde e il chip I/O sì.
- I registri del controller del Memory-Mapped I/O utilizzano lo stesso spazio di indirizzamento utilizzato dalla memoria principale.
  Il Memory-mapped I/O "spreca" parte dello spazio degli indirizzi ma non necessita di istruzioni speciali. Per le porte del dispositivo I/O della CPU sono proprio come i normali indirizzi di memoria. La CPU utilizza istruzioni simili a `MOV` per accedere ai registri del dispositivo I/O. In questo modo viene asserito l' `M/#IO` indicando l'indirizzo richiesto dalla CPU riferito alla memoria principale

Per eseguire le attività di I/O, vengono utilizzate due modalità di gestione:

- Polling
- La CPU periodicamente verifica lo stato dei task delle attività di I/O
- Interrupt-driven dove la CPU riceve un segnale di interrupt dal controller una volta che la task I/O viene completata
- La CPU riceve un interrupt dal controller (device o DMA [1]) una volta finito il task dell'I/O (con successo o in modo anomalo)
- La CPU svolge il lavoro effettivo di spostamento dei dati
- La CPU delega il lavoro a un controller DMA dedicato

1.3.3 Processori

La CPU preleva le istruzioni della memoria (esegue il **fetch**) e le esegue. La CPU si occupa poi di decodificarla per determinare il tipo e gli operandi , eseguirla e poi prelevare, decodificare ed eseguire le successive. Poichè accedere alla memoria richiede molte risorse tutte le CPU contengono all'interno dei **registri** per memorizzare variabili importanti o risultati temporanei. Oltre ai registri i computer contengono il **program counter (PC)**, contente l'indirizzo di memoria in sui si trova la successiva istruzione da seguire, dopodichè viene il PC viene aggiornato per posizionarsi sulla successiva.
Un altro registro è lo **stack pointer** che punta alla cima dello stack attuale. Un altro registro è il **program status word (PSW)** che contiene i bit di condizione, impostati da istruzioni di confronto, la prorità della CPU, la modalità (kernel o utente) e altri bit di controllo. Nelle architetture più moderne, vengono implementati più livelli di protezione, detti **protection rings**. Ogni ring aggiunge restrizioni sulle istruzioni eseguibili dalla CPU.

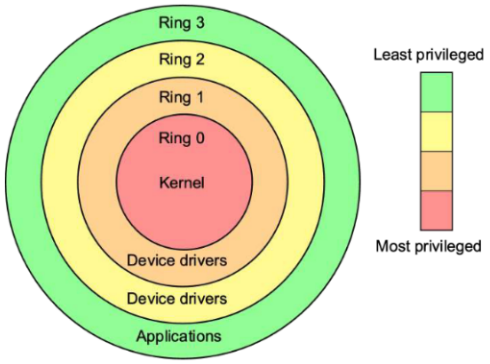

Figura 2: Protection Rings

Un modo semplice per avere una protezione alla memoria è quello di avere due registri dedicati:

- **Base** → contenente il primo address di partenza
- **Limite** → contenente l'ultimo address valido

Il sistema operativo carica i rigistri di base e limite all'avvio del programma mentre la CPU controlla che ogni indirizzo di memoria a cui fa riferimento il programma utente rientri tra i valori base e limite.

Per ottenere servizi dall'OS un programma utente deve fare una **system call (chiamata di sistema)** che entra nel kernel e richiama l'OS. Quando il lavoro è stato completato il controllo è restituito al programma utente.

L'istruzione `TRAP` cambia la modalità da utente a kernel e avvia l'OS.

- System call (software `TRAP`), ossia la richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software

[1] Direct Memory Access

- Exception (fault), ossia la gestione di errori dovuti ad eventi inattesi, svolte in modo sincrono e innescate dai software
- Interrupt, ossia il completamento di una richiesta in attesa, svolte in modo asincrono e innescate dall'hardware

Esistono sei categorie importanti di System call:

- **Gestione dei processi** include `end`, `abort`, `load`, `execute`, creazione e terminazione di processi, get/set attributi di processi, attesa, signal event, e allocazione di memoria libera. Quando un processo si interrompe o si interrompe, è necessario avviarne o riprenderne un altro. Quando i processi si arrestano in modo anomalo, potrebbe essere necessario fornire core dump e/o altri strumenti diagnostici o di ripristino
- **Gestione dei file** crea file, elimina file, apri, chiudi, leggi, scrivi, riposiziona, ottieni attributi di file e imposta attributi di file. Queste operazioni possono essere supportate anche per directory e file ordinari. L'effettiva struttura della directory può essere implementata utilizzando file ordinari sul file system o tramite altri mezzi (ne parleremo più avanti)
- **Gestione dei dispositivi** include dispositivi di richiesta, dispositivi di rilascio, lettura, scrittura, riposizionamento, recupero/impostazione degli attributi del dispositivo e collegamento o scollegamento logico dei dispositivi. I dispositivi possono essere fisici (ad es. unità disco) o virtuali/astratti (ad es. file, partizioni e dischi RAM). Alcuni sistemi rappresentano i dispositivi come file speciali nel file system, in modo che l'accesso al "file" richieda il driver di dispositivo del sistema operativo appropriato. E.g la directory `/dev` su qualsiasi sistema `UNIX`
- **Informazioni di sistema** include le chiamate per ottenere/impostare l'ora, la data, i dati di sistema e gli attributi di processo, file o dispositivo. I sistemi possono anche fornire la possibilità di eseguire il dump della memoria in qualsiasi momento. Programmi a passo singolo che interrompono l'esecuzione dopo ogni istruzione e tracciano il funzionamento dei programmi (debug).
- **Communicazione** include creazione/eliminazione di connessioni di comunicazione, invio/ricezione di messaggi, trasferimento di informazioni sullo stato e collegamento/scollegamento di dispositivi remoti. Esisono due modelli di comunicazione:
  - **scambio di messaggi**[2], il modello di trasmissione dei messaggi deve supportare le chiamate a:
    - Identificare un processo remoto e/o un host con cui comunicare
    - Stabilire una connessione tra i due processi
    - Aprire e chiudere la connessione secondo necessità
    - Trasmettere messaggi lungo la connessione
    - Attendere i messaggi in arrivo, in stato di blocco o non blocco
    - Eliminare la connessione quando non è più necessaria
  - **memoria condivisa**[3], il modello di memoria condivisa deve supportare le chiamate a:
    - Creare e accedere alla memoria condivisa tra processi (e thread)
    - Fornire meccanismi di blocco che limitano l'accesso simultaneo
    - Liberare memoria condivisa e/o allocarla dinamicamente secondo necessità

1.3.4 Protezione delle system call

Fornisce meccanismi per controllare quali utenti/processi hanno accesso a quali risorse di sistema. Le chiamate di sistema consentono di regolare i meccanismi di accesso secondo necessità. Agli utenti non privilegiati può essere concesso temporaneamente un accesso elevato autorizzazioni in circostanze specifiche. Quando un programma utente richiede l'esecuzione di una syscall tramite un'API fornita dal sistema operativo, tale richiesta viene prima convertita in linguaggio macchina, per poi venir gestita dal **System call Handler**, il quale si occuperà di salvare lo stato precedente dei registri, i quali verranno poi alterati durante l'esecuzione della syscall, per poi essere ripristinati.

[2]Nota: Più semplice e facile (in particolare per le comunicazioni tra computer) e generalmente appropriato per piccole quantità di dati

[3]Più veloce e generalmente l'approccio migliore in cui devono essere condivise grandi quantità di dati. Ideale quando la maggior parte dei processi deve leggere i dati anziché scriverli

1.3.5 Passaggio di parametri

Esistono tre metodi usati per passare parametri :

- Salvare i parametri in registri
- Salvare parametri in `block` o `table` di un'area di memoria dedicata
- Parametri inseriti nello stack dal programma e estratti dallo stack dal sistema operativo (più complessi a causa dei diversi spazi degli indirizzi)

I metodi block e stack non limitano il numero o la lunghezza dei parametri passati

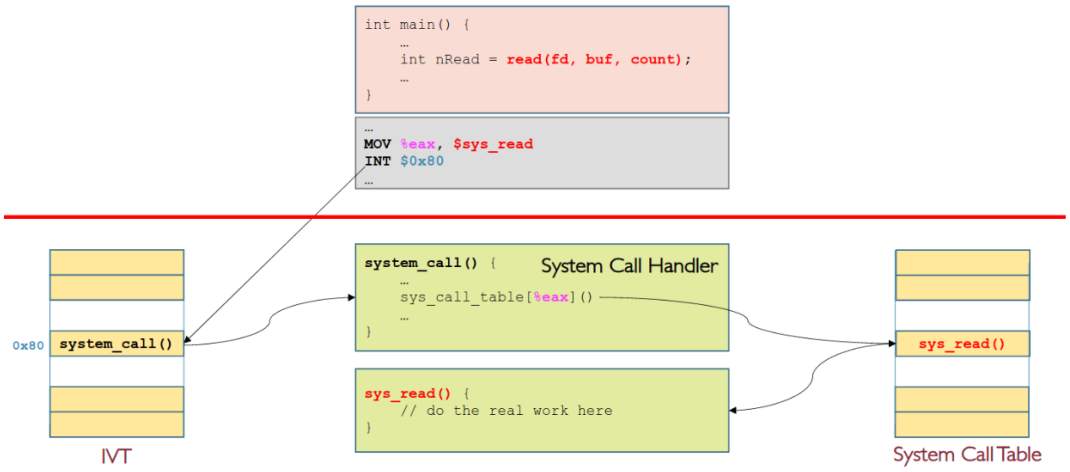

Figura 3: System call Handler

1.3.6 Timer, Istruzioni atomiche, Virtual Memory

Il timer è una funzionalità hardware per abilitare la pianificazione della CPU. Nei sistemi multi-tasking, permette alla CPU di non essere monopolizzata da processi "egoistici". Il timer genera un interrupt e ad ogni sua interruzione, lo scheduler della CPU prende il sopravvento e decide quale processo da eseguire successivamente. Gli interrupt possono verificarsi in qualsiasi momento e interferire con i processi in esecuzione e l'OS deve essere in grado di sincronizzare le attività di cooperazione, simultanee processi, garantire che brevi sequenze di istruzioni (ad es. lettura-modifica-scrittura) vengano eseguite atomicamente da disabilitare gli interrupt prima della sequenza e riabilitarli successivamente.
La virtualizzazionde della memoria è un'astrazione (dell'effettiva memoria principale fisica) e conferisce ad ogni processo l'illusione che la memoria fisica sia solo contigua spazio degli indirizzi (spazio degli indirizzi virtuali). Permette di eseguire programmi senza che vengano caricati interamente nella memoria principale. Può essere implementata sia in HW (**MMU**) che in SW (OS).

- **MMU**, associa gli indirizzi virtuali a quelli fisici tramite una tabella delle pagine gestita dal sistema operativo. Utilizza una cache denominata Translation Look-aside Buffer (TLB) con "mappature recenti" per ricerche più rapide. Il sistema operativo deve sapere quali pagine sono caricate nella memoria principale e quali su disco
- Il sistema operativo è responsabile della gestione degli spazi di indirizzi virtuali

Lo spazio degli indirizzi virtuali è tipicamente suddiviso in blocchi contigui della stessa dimensione (ad esempio, 4 KiB), chiamati pagine (**pages**). Ciascuna pagina che non sono caricate nella memoria principale vengono memorizzate su disco.

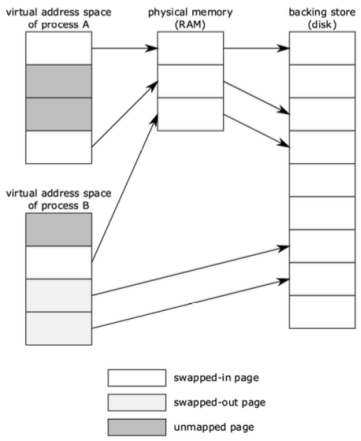

Figura 4: Virtual vs. Physical Address Space

1.4 Progettazione e implementazione del sistema operativo

La struttura interna dei diversi sistemi operativi può variare notevolmente e gli obiettivi del sistema è facilare l'usabilità rispetto al progettare/implementare. È fondamentale separare le politiche dai meccanismi della policy ovvero **cosa** sarà fatto e il meccanismo che indica **come** farlo. Il disaccoppiamento della logica della politica dal meccanismo sottostante è un principio di progettazione generale nell'informatica, in quanto migliora il sistema:

- flessibilità: l'aggiunta e la modifica delle politiche possono essere facilmente supportate
- riusabilità: i meccanismi esistenti possono essere riutilizzati per implementare nuove politiche
- stabilità: l'aggiunta di una nuova politica non destabilizza necessariamente il sistema

Le modifiche ai criteri possono essere facilmente regolate senza riscrivere il codice.

I primi sistemi operativi sviluppati in linguaggio assembly, e uno dei vantaggi era il controllo diretto sull'HW (alta efficienza) mentre uno svantaggio può essere il legame con uno specifico HW (bassa portabilità). Oggi abbiamo un misto di linguaggi e ai livelli più bassi troveremo assembly il corpo principale in C e i programmi di sistema in C, C++, linguaggi di scripting come PERL, Python, ecc.
Il sistema operativo dovrebbe essere suddiviso in sottosistemi separati, ciascuno con compiti, input/output e caratteristiche prestazionali accuratamente definiti. Esistono vari modi di strutturare un sistema operativo:

- **Struttura semplice**, dove non vi è alcun sottosistema e non vi è separazione tra kernel e user mode (esempio: il sistema MS-DOS). Semplice da implementare ma estremamente insicuro e poco rigido.
- **Struttura a Kernel Monolitico**, dove l'intero sistema operativo opera in kernel mode e solo i software utente lavorano in user mode (esempio: il sistema UNIX). Semplice da implementare ed efficiente, ma ancora poco sicuro e rigido
- **Struttura a livelli** (come MULTICS) dove l'OS è suddiviso in N livelli ed ogni livello L usa funzionalità implementate dal livello L_1 ed espone nuove funzionalità al livello L + 1. Per via della struttura a livelli, il sistema è molto modulare, portabile e semplice da debuggare, rendendo tuttavia più complessa per la comunicazione tra di essi.
- **Struttura a Microkernel**, dove il kernel contiene solo le funzionalità di base, mentre tutte le altre funzionalità dell'OS e i programmi utente vengono eseguiti in user mode. Tale struttura porta ad una maggiore sicurezza, affidabilità ed estensibilità, ma anche ad un'efficienza ridotta.
- **Struttura a Moduli del Kernel caricabili (LKM)**, dove l'OS utilizza dei moduli tramite cui accedere alle funzionalità del kernel
- **Sistema a Kernel Ibrido**, dove viene utilizzato un approccio intermedio al kernel monolitico e al microkernel, ottenendo i vantaggi di entrambi gli approcci

2 Gestione dei Processi e thread

Un programma è un file eseguibile che risiede nella memoria persistente (ad es. disco) e contiene solo l'insieme di istruzioni necessarie per eseguire un lavoro specifico. Un processo è l'astrazione del sistema operativo di un programma in esecuzione (unità di esecuzione). Il processo è dinamico, mentre un programma è statico (solo codice e dati). Diversi processi possono eseguire lo stesso programma (ad esempio, multiple istanze di Google Chrome) ma ognuna ha il proprio stato. Un processo esegue un'istruzione alla volta, in sequenza.

Il sistema operativo fornisce la stessa quantità di spazio di indirizzi virtuali a ciascun processo. Lo spazio degli indirizzi virtuali è un'astrazione dello spazio degli indirizzi della memoria fisica. L'intervallo di indirizzi virtuali validi che un processo può generare dipende dalla macchina. Avremo perciò:

- **Text** contenente le istruzioni dell'eseguibile
- **Data** Variabili globali e statiche inizializzate
- **BSS** variabile globale e statica (non inizializzata o inizializzata a 0)
- **Stack** Struttura LIFO utilizzata per memorizzare tutti i dati necessari a una chiamata di funzione (stack frame)
- **Heap** usata per l'allocazione dinamica

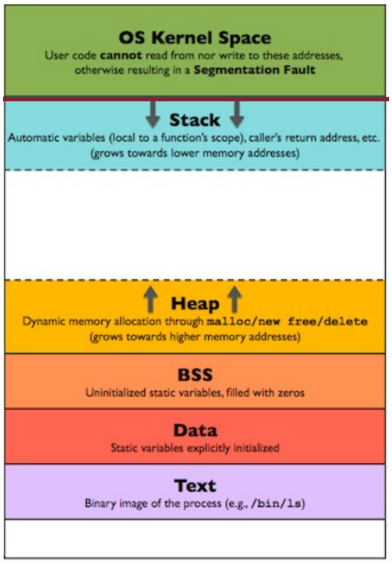

Figura 5: Virtual Address Space Layout

Sullo stack sono definite due operazioni: **push** e **pop**. Il push viene usato per inserire degli elementi, mentre pop per rimuoverli. Usa un registro dedicato (e.g `esp`) il cui contenuto è l'indirizzo nella memoria principale della parte superiore dello stack. La memoria dello stack cresce convenzionalmente dall'alto verso il basso, cioè dagli indirizzi di memoria più alti a quelli più bassi. Ogni funzione utilizza una porzione dello stack, chiamata **stack frame** e in ogni momento possono esistere contemporaneamente più stack frame, a causa di diverse chiamate di funzioni nidificate, ma solo una è attiva. Lo stack frame per ogni funzione è diviso in tre parti:

- parametri della funzione + indirizzo di ritorno
- back-pointer allo stack frame precedente
- variabili locali

Il primo è impostato dal chiamante, il secondo e il terzo vengono impostati dal chiamato. Il puntatore `esp` viene sempre aggiornato man mano che lo stack cresce ed è difficile per il chiamato accedere ai parametri effettivi senza un riferimento fisso nello stack. Per risolvere questo problema invece di utilizzare un singolo puntatore in cima allo stack usa un puntatore aggiuntivo alla parte inferiore (base) dello stack (`%ebp`) lascia che `esp` sia libero di cambiare tra diverse chiamate di funzione, mentre tiene `%ebp` fissato all'interno di ogni stack frame.

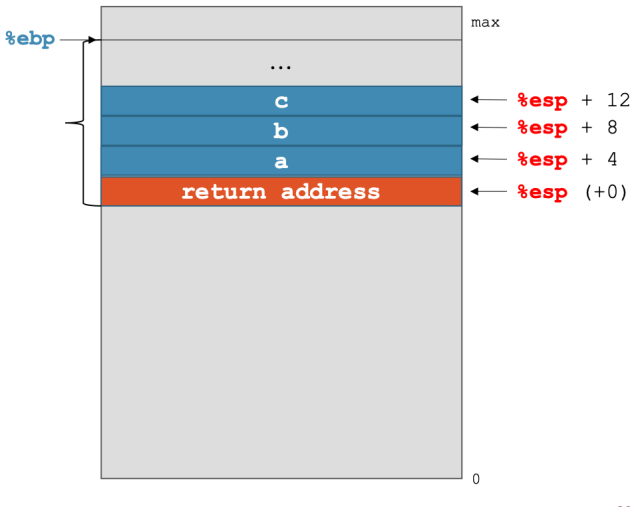

Figura 6: Function Parameters + Return

2.1 Stato di esecuzione del processo

In ogni momento un processo può trovarsi in uno dei seguenti cinque stati:

- **New**, il processo appena avviato
- **Ready**, il processo è pronto per essere eseguito ma attende di essere programmato sulla CPU
- **Running**, il processo sta effettivamente eseguendo istruzioni sulla CPU
- **Waiting**, il processo è sospeso in attesa che una risorsa sia disponibile o un evento da completare/verificare (ad es. input da tastiera, accesso al disco, timer, ecc.)
- **Terminated**, il processo è terminato e il sistema operativo può distruggerlo

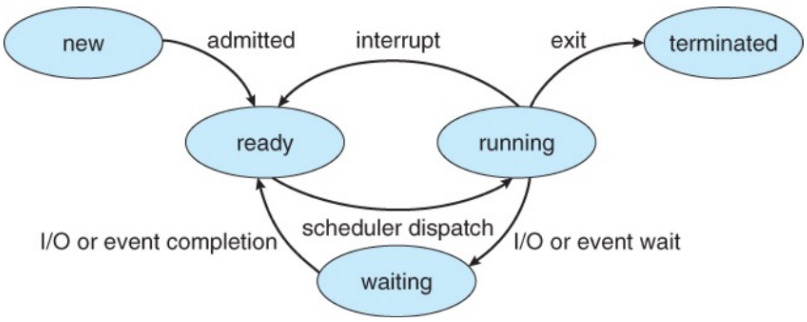

Figura 7: Process Execution State Diagram

Durante l'esecuzione, il processo passa da uno stato all'altro in base alle azioni del programma (ad esempio, chiamate di sistema) come azioni del sistema operativo (ad es. pianificazione) o azioni esterne (ad es. interruzioni).

La maggior parte delle chiamate di sistema (ad esempio quelle di I/O) sono bloccanti. Il processo chiamante (spazio utente) non può fare nulla fino al ritorno della chiamata di sistema.
Il sistema operativo (spazio del kernel) si occupa di:

- impostare il processo corrente in uno stato di attesa (ovvero, in attesa del ritorno della chiamata di sistema)
- pianifica un diverso processo pronto per evitare che la CPU sia inattiva

Una volta che la chiamata di sistema ritorna, il processo precedentemente bloccato è pronto per essere schedulato nuovamente per l'esecuzione.[4]
Lo stato del processo è costituito da:

- il codice del programma in esecuzione
- i dati statici del programma in esecuzione
- il program counter (PC) che indica la prossima istruzione da eseguire
- Registri CPU
- la catena di chiamate del programma (stack) insieme ai puntatori di frame e stack
- lo spazio per l'allocazione dinamica della memoria (heap) e al suo puntatore, le risorse in uso (ad es. file aperti)
- lo stato di esecuzione del processo (pronto, in esecuzione, ecc.)

2.2 Process Control Block (PCB)

Il **PCB** (o detto anche Tabella dei Processi), è la struttura dati principale utilizzata dal sistema operativo per tenere traccia di qualsiasi processo. Il PCB tiene traccia dello stato di esecuzione e della posizione di un processo. Il sistema operativo assegna un nuovo PCB alla creazione di un processo e lo inserisce in una coda di stato e viene deallocato non appena termina il processo associato.
Associata a ogni classe di I/O c'è una posizione (solitamente in una collocazione fissa vicino alla base della memoria) chiamata **vettore di interrupt**[5]. Contiene l'indirizzo della procedura di servizio dell'interrupt. Ad ogni interrupt il computer salta all'indirizzo specificato nel vettore dell'interrupt. Il PCB contiene:

- Stato del processo pronto, in attesa, in esecuzione, ecc.
- Numero di processo (ovvero identificatore univoco)
- Program Counter (PC) + Stack Pointer (SP) + registri di uso generale
- Informazioni sulla pianificazione della CPU priorità e puntatori alle code di stato
- Informazioni sulla gestione della memoria tabelle delle pagine
- Informazioni sull'account, ossia il tempo utilizzato dalla CPU del kernel e dell'utente, stato I/O del proprietario
- Elenco dei file aperti

[4]NOTA: l'intero sistema non è bloccato, solo il processo che ha richiesto la chiamata bloccata è!

[5]In informatica, un interrupt vector (vettore delle interruzioni) è un indirizzo di memoria del gestore di interrupt, oppure un indice ad un array, chiamato interrupt vector table, il quale può essere implementato tramite una dispatch table. La tabella degli interrupt vector contiene gli indirizzi di memoria dei gestori di interrupt. Quando si genera una interruzione, il processore salva il suo stato di esecuzione con il context switch, ed inizia l'esecuzione del gestore di interruzione all'interrupt vector (questo procedimento avviene quando l'interruzione ha carattere sincrono). Infatti vi è una wait instruction (istruzione d'attesa) che obbliga la CPU ad effettuare cicli vuoti fino a che non arriva il prossimo interrupt. Nel caso in cui si dovesse verificare una interruzione asincrona il programma provvede a lanciare un interrupt con la richiesta di I/O e nell'attesa del dato continua ad effettuare operazioni logico-aritmetiche. Quando arriverà nel momento in cui gli servirà il risultato si fermerà e inizierà ad aspettare. L'I/O asincrono serve a far sì che alcuni programmi possano anticipare la richiesta di un dato così nel caso in cui esso dovesse servire lo possono già utilizzare. Fonte: Wikipedia

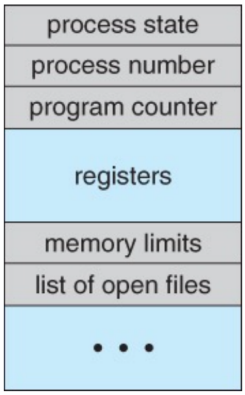

Figura 8: Process Execution State Diagram

2.3 Creazione di processi

I processi possono creare altri processi tramite specifiche chiamate di sistema. Il processo creatore è chiamato genitore del nuovo processo, chiamato figlio. Il genitore condivide risorse e privilegi con i suoi figli. Un genitore può aspettare che un figlio finisca o continuare in parallelo.
A ogni processo viene assegnato un identificatore intero (noto anche come identificatore di processo o PID) e per ogni processo viene memorizzato anche il PID genitore (PPID).
Sui tipici sistemi UNIX lo scheduler del processo è denominato `sched` e riceve il PID 0. La prima cosa che fa all'avvio del sistema è lanciare `init`, che dà a quel processo il PID 1. Successivamente `init` avvia tutti i daemons di sistema e i login degli utenti e diventa il genitore ultimo di tutti gli altri processi. I processi vengono creati attraverso la chiamata di sistema `fork()`.

Tutti i processi sono uguali. L'unico indizio di gerarchia dei processi è che quando viene creato un processo, al genitore viene dato uno speciale gettone o token (detto **handle**) che può controllare il figlio invalidando la gerarchia.

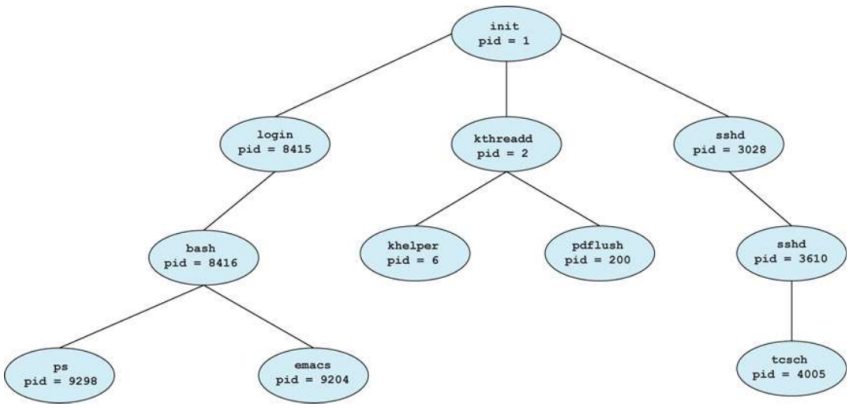

Figura 9: Process Creation: UNIX/Linux

2.3.1 Risorse del padre vs figlio

Abbiamo due possibilità per lo spazio degli indirizzi del figlio rispetto al genitore:

- Il bambino può essere un duplicato esatto del genitore, condividendo lo stesso programma e gli stessi segmenti di dati in memoria
- Ciascuno avrà il proprio PCB, inclusi program counter, registri e PID
- Questo è il comportamento della chiamata di sistema `fork()` in UNIX
- Il processo figlio può avere un nuovo programma caricato nel suo spazio degli indirizzi, con tutti i nuovi segmenti di codice e dati
- In Windows le chiamate di sistema saranno create con il comando `spawn()`
- I sistemi UNIX implementano ciò come secondo passaggio, utilizzando la chiamata di sistema `exec`.

2.3.2 Esecuzione del padre vs figlio

Abbiamo due opzioni per il processo genitore dopo aver creato il figlio:

- Attendere che il processo figlio termini prima di procedere emettendo un `wait` chiamata di sistema, per un figlio specifico o per qualsiasi figlio
- Oppure eseguire contemporaneamente al figlio, continuando l'elaborazione senza essere bloccato (quando una shell UNIX esegue un processo come attività in background utilizzando "&")

Consideriamo ora questo esempio:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid;
    /* fork a child process */
    pid = \texttt{fork()};
    if (pid < 0) {
        /* if the returned PID is < 0, an error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
    /* execute child process code */
        execlp("/bin/ls", "ls", NULL);
    }
    else {
    /* execute parent process code */
        ...
        /* wait for child to terminate */
        wait(NULL)
        printf("Child terminated");
        exit(0);
    }
}
```

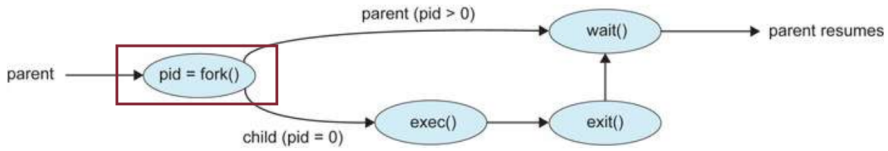

Figura 10: Albero decisionale del primo esempio

Consideriamo ora quest'altro esempio:

```
    int pid = \texttt{fork()};
    if(pid == 0) {        // A's child (B)
        pid = \texttt{fork()};

    if(pid == 0) {        // B's child (C)
        ...
        execlp(...);
    }
    else { // B
        ...
    }
}
else { // A
    ...
}
```

Figura 11: Albero decisionale del secondo esempio

Consideriamo ora questo esempio:

```
int pid = \texttt{fork()};
if(pid == 0) {              // A's child (B)
    ...
    execlp(...);
}
else { // A
    pid = \texttt{fork()};
    if(pid == 0) {          // A's child (C)
        ...
        execlp(...);
    }
}
```

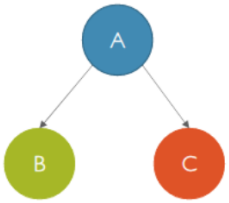

Figura 12: Albero decisionale del secondo esempio

Consideriamo ora quest'altro esempio:

```
for(int i=0;i<n;i++) {
    if(\texttt{fork()} == 0) { // A0's child
        ...
        execlp(...);
    }
}
// back in the parent A0
// wait for all children to terminate

for(int i=0;i<n;i++) {
    wait(NULL);
}
```

Figura 13: Albero decisionale del secondo esempio

Riepilogo delle chiamate di sistema viste:

- `fork()` genera un nuovo processo figlio come copia esatta del genitore
- `execlp` sostituisce il programma del processo corrente con il inserire il programma denominato
- `sleep` sospende l'esecuzione per un certo numero di secondi
- `wait`/`waitpid` attende che uno o più processi specifici terminino l'esecuzione

2.4 Terminazione di processi

I processi possono richiedere la propria terminazione effettuando la chiamata di sistema `exit`, tipicamente restituendo un *int*. Questo *int* viene passato al genitore se sta eseguendo un'attesa. Solitamente è 0 in caso di completamento con successo e un diverso da zero in caso di problemi.
I processi possono anche essere terminati dal sistema per una serie di motivi:

- L'incapacità del sistema di fornire le risorse di sistema necessarie
- In risposta a un comando `kill` o ad un altro interrupt di processo non gestito
- Uscita normale (condizione volontaria)
- Uscita su errore (condizione volontaria)
- Terminato da un altro processo (condizione involontaria)

Un genitore può uccidere i suoi figli se il compito loro assegnato non è più necessario, il sistema però può consentire (o meno) al figlio di continuare senza un genitore. Sui sistemi UNIX, i processi orfani sono generalmente ereditati da `init`, che poi li uccide.

Quando un processo termina, tutte le sue risorse di sistema vengono liberate, i file aperti svuotati e chiusi, ecc. Lo stato di terminazione del processo dei tempi di esecuzione vengono restituiti al genitore se è in attesa che il figlio termini o `init` se il processo diventa orfano.

I processi sono detti **zombie** se stanno tentando di terminare ma non possono perché il loro genitore non li sta aspettando oppure ereditati da `init` come orfani e uccisi.

2.5 Scheduling dei processi

Gli obiettivi principali del sistema di schedulazione dei processi:

- mantenere la CPU sempre occupata
- fornire tempi di risposta "accettabili" per tutti i programmi, in particolare per quelli interattivi

Lo scheduler del processo deve soddisfare questi obiettivi implementando politiche adeguate per lo scambio dei processi dentro e fuori la CPU. Questi obiettivi però possono essere contrastanti, e ogni volta che il sistema operativo interviene per scambiare i processi, impiega tempo sulla CPU per farlo, che viene quindi "perso" dal fare qualsiasi lavoro produttivo utile.

Il sistema operativo mantiene i PCB di tutti i processi nelle code di stato. C'è una coda per ciascuno dei cinque stati in cui può trovarsi un processo e una per ogni dispositivo I/O (dove i processi attendono che un dispositivo diventi disponibile o consegni i dati). Quando il sistema operativo cambia lo stato di un processo, il PCB viene scollegato dalla coda corrente e spostato in quella nuova. Il sistema operativo può utilizzare criteri diversi per gestire ciascuna coda di stato.

La coda di esecuzione è vincolata dal numero di core disponibili sul sistema. Ogni volta, è possibile eseguire un solo processo su una CPU. Non esiste un limite teorico al numero di processi negli stati `new/ready/waiting/terminated`.

Esisono diversi scheduler (che in un sistema efficiente utilizza un mix tra questi):

- Uno **scheduler a lungo termine** viene eseguito raramente ed è tipico di un sistema batch o di un sistema molto carico
- Uno **scheduler a breve termine** viene eseguito molto frequentemente (circa ogni 100 millisecondi) e deve sostituire molto rapidamente un processo dalla CPU e inserirne un altro
- Alcuni sistemi utilizzano anche uno **scheduler a medio termine**: quando i carichi del sistema diventano elevati, questo scheduler consente ai lavori più piccoli e più veloci di terminare rapidamente e liberare il sistema

2.5.1 Il Context Switch

Il **Context Switch** procedura utilizzata dalla CPU per sospendere il processo attualmente in esecuzione al fine di eseguirne uno pronto. È un'operazione molto costosa in quanto l'arresto del processo in corso implica il salvataggio di tutto il suo stato interno (PC, SP, altri registri, ecc.) al suo PCB e l'avvio di un processo pronto consiste nel caricare tutto il suo stato interno (PC, SP, altri registri, ecc.) dal suo PCB.
Il Context Switch si usa quando ci sono delle chiamate `TRAP`[6] come system call, eccezioni, o interruzioni hardware. Ogni volta che arriva una `TRAP` la CPU ha il compito di creare uno stato di salvataggio dello stato corrente dei processi attivi. Passare alla modalità kernel per gestire l'interrupt ed eseguire un ripristino dello stato del processo interrotto.
Per evitare che i processi legati alla CPU monopolizzino la CPU, anche il Context Switch viene attivato tramite interrupt del timer hardware (**slice**). Lo **slice** ha quantità massima di tempo tra due cambi di contesto per garantire che si verifichi almeno un cambio di contesto. Può accadere più frequentemente di così (ad esempio, a causa di richieste di I/O). Lo slice è facilmente implementabile in hardware tramite interrupt timer. E' un meccanismo utilizzato dai moderni sistemi operativi multitasking time-sharing per aumentare la reattività del sistema (pseudo-parallelismo). [7]

Il tempo nel Context Switch

Il tempo impiegato per completare un cambio di contesto è solo tempo di CPU sprecato. Un intervallo di tempo inferiore comporta cambi di contesto più frequenti. Un intervallo di tempo maggiore comporta cambi di contesto meno frequenti. Ha anche il compito di minimizzare lo spreco di tempo della CPU, massimizzando quindi l'utilizzo e la reattività della CPU.

2.6 Communicazione dei processi

I processi possono essere **indipendenti** o **cooperanti**:

- Processi **indipendenti** operano contemporaneamente su un sistema e possono né influenzare né essere influenzati da altri processi
- I processi **cooperanti** possono influenzare o essere influenzati da altri processi al fine di raggiungere un compito comune

Possono esserci diversi processi che richiedono l'accesso allo stesso file (ad es pipeline). Abbiamo bisogno di questa cooperazione tra processi e ci sono diverse caratteristiche importanti. Abbiamo la **velocità** di calcolo è un problema che può essere risolta più velocemente se può essere suddiviso in sotto-attività da risolvere simultaneamente. La **modularità** definisce l'architettura più efficiente può essere quella di scomporre un sistema in moduli cooperanti. Infine abbiamo la **convenienza** infatti anche un singolo utente può essere multitasking, come modificare, compilare, stampare ed eseguire lo stesso codice in finestre diverse. Ci sono due modi possibili per comunicare tra processi cooperanti:

- **Memoria condivisa**[8]
  - Più complicato da configurare e non funziona altrettanto bene su più computer
  - Più veloce una volta configurato, poiché non sono necessarie chiamate di sistema
  - Preferibile quando (una grande quantità) di informazioni deve essere condivisa sullo stesso computer
- **Scambio di messaggi**
  - Più semplice da configurare e funziona bene su più computer
  - Più lento in quanto richiede chiamate di sistema (syscall) per ogni trasferimento di messaggi

[6] richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software

[7] Diverso invece quando si parla di **multiprocessore** che hanno due o più CPU che condividono la stessa memoria fisica, vero parallelismo hardware

[8] Per far sì che un processo possa richiedere un'area di memoria da condividere con altri processi c'è, ovviamente, necessità di effettuare una chiamata di sistema (a livello utente, non sarebbe possibile completare quest'operazione). Tuttavia, una volta "pagato questo prezzo iniziale", le interazioni successive tra processi che condividono un'area di memoria può avvenire senza ulteriori chiamate di sistema. **Fonte: il Prof**

  - Preferibile quando la quantità e/o la frequenza dei trasferimenti di dati è piccola o quando sono multipli i computer sono coinvolti

La memoria da condividere è inizialmente all'interno dello spazio degli indirizzi di un particolare processo e bisogna effettuare chiamate di sistema per rendere quella memoria pubblicamente disponibile ad altri processi. Altri processi devono effettuare le proprie chiamate di sistema per collegare la memoria condivisa al proprio spazio degli indirizzi.
Il **Message Passing Systems** deve supportare almeno chiamate di sistema per l'invio e la ricezione di messaggi e stabilire un collegamento di comunicazione tra i cooperanti processi prima che i messaggi possano essere inviati.

Esistono però tre problemi da risolvere nello scambio di messaggi:

- Comunicazione diretta o indiretta (es. naming)
  - Nella **Comunicazione diretta** il mittente deve conoscere il nome del destinatario a cui desidera inviare un messaggio. Deve conoscere anche il collegamento uno a uno tra ogni coppia mittente-destinatario per la comunicazione simmetrica e il destinatario deve conoscere anche il nome del mittente.
  - La **Comunicazione indiretta** utilizza mailbox o porte condivise dove più processi possono condividere la stessa mailbox o porta. Solo un processo può leggere un determinato messaggio in una mailbox. Il sistema operativo deve fornire chiamate di sistema per creare ed eliminare mailbox e per inviare e ricevere messaggi da/per mailbox.
- Comunicazione sincrona o asincrona
- Buffering automatico o esplicito

Scegliere se utilizzare o meno una queue per i messaggi:

- **Zero capacity**, dove i messaggi non possono essere salvati in una queue, dunque il mittente rimane bloccato finché il destinatario non riceve il messaggio.
- **Bounded capacity**, dove vi è una queue di capienza predefinita, permettendo ai mittenti di non bloccarsi a meno che la queue non sia piena.
- **Unbounded capacity**, dove la queue ha teoricamente capienza infinita, implicando che i mittenti non debbano mai bloccarsi.

2.7 Scheduling dei processi

Definiamo come CPU burst lo stato in cui un processo è eseguito dalla CPU, mentre definiamo come I/O burst lo stato in cui un processo è in attesa che i dati vengano trasferiti dentro o fuori dal sistema. In generale, ogni processo alterna costantemente tra CPU burst e I/O burst.
Lo scheduling dei processi è una **politica** che stabilisce quali processi devono essere eseguiti dalla CPU. Lo scheduler della CPU è un processo che, a seconda di una policy di scheduling stabilita, si occupa di selezionare un processo dalla ready queue da eseguire in ogni momento in cui la CPU è inattiva. La struttura dati utilizzata per la ready queue e l'algoritmo usato per selezionare il processo successivo è basato su una queue FIFO (First In, First Out)

- Concetti di base della pianificazione. Quasi tutti i programmi hanno alcuni cicli alternati di calcoli della CPU e attesa di I/O. Anche un semplice recupero dalla memoria principale richiede molto tempo rispetto alla velocità della CPU. (Consideriamo per ora la multiprogrammazione sui sistemi con un unico processore)
- Algoritmi di pianificazione
- Concetti avanzati di pianificazione.

Ogni volta che la CPU diventa inattiva lo **scheduler** della CPU seleziona un altro processo dalla coda pronta per l'esecuzione successiva. La struttura dei dati per la coda pronta e l'algoritmo utilizzato per selezionare il processo successivo non sono però necessariamente basati su una coda FIFO. Le **decisioni di pianificazione** della CPU avvengono in una delle quattro condizioni:

1. Quando un processo passa dallo stato in **esecuzione** allo stato di **attesa**
2. Quando un processo passa dallo stato in **esecuzione** allo stato **pronto**
3. Quando un processo passa dallo stato di **attesa** allo stato **pronto**
4. Quando un processo viene **creato** o **terminato**

Nel 1° e il 4° caso è necessario selezionare un nuovo processo, mentre nel 2° e 3° caso si continua con il processo corrente o bisogna selezionarne uno nuovo.

2.8 Scheduling Preemptive e Non-Preemptive

Un algoritmo di scheduling **non-preemptive** assegna la CPU a un processo e non lo interrompe finché non termina o finché non rilascia volontariamente la CPU. Ciò significa che un processo che richiede una quantità considerevole di tempo della CPU può bloccare altri processi che sono in attesa di esecuzione.

Un algoritmo di scheduling **preemptive**[9], al contrario, consente al sistema operativo di interrompere l'esecuzione di un processo in qualsiasi momento per consentire l'esecuzione di un altro processo in coda. Questo significa che i processi con un tempo di esecuzione più breve possono essere eseguiti più rapidamente, anche se ci sono altri processi in attesa di esecuzione. L'algoritmo di scheduling preemptive è anche chiamato round-robin.

2.8.1 Problemi

Preemption potrebbe causare problemi se si verifica mentre il kernel è impegnato nell'implementazione di una system call (ad esempio, l'aggiornamento dei dati critici del kernel strutture) oppure due processi condividono dati, uno può essere interrotto durante l'aggiornamento di strutture dati condivise. Ci sono però possibili **soluzioni**, si può fare in modo che il processo attenda finché la chiamata di sistema non è stata completata o bloccata prima di concedere la preemption (problematico per i sistemi in tempo reale, poiché la risposta in tempo reale non può più essere garantita) oppure disabilitare gli interrupt prima di entrare nella sezione del codice critico e riabilitarli subito dopo (dovrebbe essere fatto solo in rare situazioni e solo su parti di codice molto brevi che finiranno rapidamente).

2.9 Il Dispatcher

Il dispatcher è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler. Ha il compito di:

- Cambio di contesto
- Passaggio alla modalità utente
- Saltare alla posizione corretta nel programma appena caricato
- Il dispatcher viene eseguito su ogni cambio di contesto quindi il tempo che esso consuma (latenza di invio) deve essere il più breve possibile

2.10 Definizioni utili

[10]

- Orario di arrivo: ora in cui il processo arriva nella coda pronta
- Tempo di completamento (Arrival Time): momento in cui il processo completa la sua esecuzione
- Burst Time: tempo richiesto da un processo per l'esecuzione della CPU
- TurnaroundTime: differenza di tempo tra il completamento e l'orario di arrivo
- Waiting Time: differenza di tempo tra il tempo di turnaround e il burst time

[9] Nota: uno scheduler preemptive viene comunque attivato dalle condizioni 1 e 4, poiché in tali casi deve comunque essere selezionato un processo.

[10] NOTA: Il tempo di attesa I/O non è considerato qui!

$$T^{arrival} = \text{arrival time}$$

$$T^{completion} = \text{completion time}$$

$$T^{burst} = \text{burst time}$$

$$T^{turnaround} = \text{tournaround time} = T^{completion} - T^{arrival}$$

$$T^{waiting} = \text{waiting time} = T^{turnaround} - T^{burst}$$

Figura 14

da pag 49 del pdf slide 6

3 Sincronizzazione tra Processi/Thread

4 Gestione della Memoria

5 Gestione dei Sistemi di I/O

6 File System

7 Advanced Topics