Scheduler.

All'interno di un sistema multi-process schedula i vari processi da mandare in esecuzione, può avere politica diversa (round robin, x necessità etc.)
Dal punto di vista dello scheduler, se prendo un asse temporale lo scheduler rende tutto come una sequenza di istruzioni composta delle istruzioni dei diversi processi in esecuzione.

Es. P1 (ist 1,1 - ist 2,1 - ist 3,1) P2 (ist1,2- ist2,2- ist3,2) P3 (ist1,3 - ist2,3- ist3,3)
Lo scheduler è colui che dirà:
ist 1,1 - ist 1,2 - ist 2,2 - ist2,1 - ist3,1 - ….

L'unica cosa che viene preservata è l'ordine locale delle istruzioni di ogni processo. Ma non sapremo mai tra le varie istruzioni dei vari processi quale verrà eseguita prima.

Scheduler: fa lo shuffling delle istruzioni dei vari processi in un sistema multi-processo.
Scheduler Fair - da accesso a tutti i processi (primo o poi)
Scheduler Fair , Round Robin - è un particolare politica dello scheduler Fair che da accesso a tutti a rotazione per quanti di tempo
Scheduler Fair, a priorità - altra politica che potrebbe privilegiare quelli con più priorità, ma cmq TUTTI vengono eseguiti prima o poi.

Processo.

Un elemento che include:

- del codice (probabilmente condiviso dai diversi processi che lo utilizzano)
- Un set di dati
- Un numero di attributi che descrivono lo stato del processo:
    - identificatore
    - stato
    - priorità
    - program counter
    - puntatori in memoria
    - dati di contesto (chi è amministratore etc.)
    - informazioni I/O
    - **

- Questi elementi compongono il Process Control Block

Creazione nuovo processo.
Il processo si crea con delle chiamate al sistema:
fork() - creazione di un'istanza del processo, ritorna -1 se insuccesso (es. non ci sono abb risorse x creare un nuovo processo), ritorna 0 lato figlio se tutto va bene e ritorna l'identificatore del figlio dal lato padre se tutto va bene

Alla creazione di un processo (figlio) da parte di un'altro processo (padre) i due vengono eseguiti contemporanemente.
Quando il padre crea il figlio si effettua una duplicazione del padre (con una copia della memoria, registri CPU, risorse in uso dal padre), poi il figlio da entità indipendente potrà cambiare i suoi dati e il suo PCB. I due hanno PCB Data Text separati, ma possono condividere risorse e file.

All'avvio del sistema operativo vi è un solo processo padre che poi crea pian piano gli altri processi, che controllano le periferiche, la gestione etc.

Il processo padre può decidere di aspettare che un figlio muoia : questo avviene quando un padre avvia un processo che deve soddisfare una richiesta e basta. E quindi il padre si mette in attesa della restituzione della richiesta e dopodiché il processo figlio muore e automaticamente gli passa al padre il valore risultato della sua morte. La morte è il metodo per ricevere un risultato da un figlio che deve unicamente soddisfare una richiesta.

<u>Funzionamento:</u>

Padre:
ret = fork(); —> Nasce il figlio e ProgramCounter +1
switch(ret){
case -1: error, exit(1):
case 0 : codice x il figlio
case default : codice x il genitore, wait(0);

Figlio: copia codice del padre e parte esecuzione da istr successiva alla fork ():
ret = fork();
switch(ret){ —> parte esecuzione da qui
case -1: error, exit(1):

case 0 : codice x il figlio
case default : codice x il genitore, wait(0);

L'unica cosa che differisce dall'esecuzione dei due è il return della fork , lato figlio = 0 , lato padre =PID figlio.

Entrambe i processi sono ora attivi e lo scheduler sarà colui che sceglierà chi mandare in esecuzione tra i due e quindi chi effettuerà prima lo switch tra i due.

Il wait(0) del padre mette in attesa il padre fintanto che tutti i figli siano morti.
Se come argomento gli passo (0) è un wait non selettivo quindi attenderà che tutti i figli siano morti, posso anche passargli un PID con waitpid(PID) e quindi aspettare che solo quel processo sia morto.

L'exit(0) del figlio manda messaggio asincrono al padre e sblocca il wait del padre per permettergli di continuare con l'esecuzione.
Se exit(0) viene eseguito prima ancora che il padre entri nello wait() , il figlio passa ad uno stato zombie per rimanere vivo fintanto che il padre eseguirà la wait ed il sistema operativo controllerà se i processi figli sono in stato zombie allora il padre viene sbloccato.
In stato zombie le uniche cose del processo che vengono preservate è lo Stato, il PID e se richiesto un valore di risultato. Allo sblocco del padre anch'esse vengono ripulite.

<u>Funzionamento exit():</u>
Si chiudono tutti i file aperti, si rialloca la memoria utilizzata dal processo ma si salvano i risultati delle operazioni.
Si controlla se padre vivo = si mantiene risultato fino alla wait del padre, ed il figlio entra in stato zombie
Si controlla se padre non vivo =il figlio, se c'e' wait gli rimanda messeggio dell'exit, se non c'e' wait viene direttamente rimosso.

Fork VS Thread:

E' più vantaggioso usare i thread quando i task da fare all'interno di un'applicazione sono parallelizzabili, quando sono indipendenti tra loro = multi-thread.

Due tipici modi per creare programmi basati su thread:

- Master - Slave: manager assegna task a thread figli (modello che vale anche x processi)
- Pipeline : piccoli pezzettini di una grande operazione fatti in parallelo ove possibile
- Se i figli devono avere zone di memoria private allora più opportuna un multi-processo, se invece non è necessario avere variabili locali ai singoli processi e posso invece condividere le zone di memoria allora è più opportuno usare i thread in unico processo.

Shared Memory Model:
Esiste un processo padre, che genera i vari thread i quali hanno tutti accesso agli stessi dati del processo che si trovano nella sezione dati in memoria.
Vantaggi come velocità, svantaggio per la sincronizzazione perché ognuno entra legge e scrive quando vuole.

Thread-Safeness:
Creare un applicazione basata su thread i quali aggiornano i dati condivisi, c'e' quindi sincronizzazione che fa perdere punti di performance ma garantisce la sicurezza.

Operazione atomica: operazione parte e termina