Quando più processi accedono contemporaneamente alla stessa risorsa fisica o logica, si ha la concorrenza. Se l'esito delle operazioni dipende dall'ordine in cui sono eseguite, si ha una corsa critica, e ciò può causare inconsistenza nella risorsa. Una porzione di codice che causa corse critiche se eseguita in modo concorrente è detta sezione critica. Ciò va evitato con meccanismi e politiche di sincronizzazione.
Una soluzione al problema delle sezioni critiche deve soddisfare 3 condizioni:

- Mutua esclusione: un solo processo per volta deve eseguire la sua sezione critica
- Progresso: la competizione per entrare nella sezione critica deve avvenire tra i processi che non sono nella loro sezione critica, e la competizione non deve durare indefinitamente
- Attesa limitata: un processo non deve essere costretto ad attendere indefinitamente per entrare nella sua sezione critica

Vediamo ora alcune soluzioni.
Variabili di turno:
Si tratta di un approccio a livello di istruzioni. Una variabile di turno è una variabile condivisa tra i processi che stabilisce quale può usare una risorsa.
Si possono usare per risolvere il problema delle sezioni critiche. Esempio per 2 processi/thread:
si usano 2 flag inizializzati a false che rappresentano quando un processo è o vuolr entrare nella propria sezione critica; e una variabile di turno, inizializzata ad uno dei 2 processi. Quando un processo vuole entrare nella propria sezione critica, imposta il turno sull'altro e il proprio flag a true, e attende se e solo se il turno è quello dell'altro, e questi ha il flag a true; al rilascio imposterà il proprio flag a false. Grazie alla doppia condizione, se entrambi tentano di entrare contemporaneamente, solo uno dei 2 ci riuscirà, a seconda del valore finale assunto della variabile di turno, che decide chi entrerà per primo. In questo modo si rispettano le 3 condizioni e si risolve il problema della sezione critica.
Questo metodo all'aumentare dei processi scala male: serve infatti modificare il codice o scrivere codice che si adatti.
Variabili di lock:
E' un altro approccio a livello di istruzioni, ma non sono i processi ad alternarsi, bensì la risorsa ad avere una variabile che dica se è disponibile (0) oppure in uso (1). Questo metodo è indipendente dal numero di processi. Quando un processo vuole utilizzarle la risorsa, deve ottenere un lock in questo modo:

- Disabilitare gli interrupt
- Leggere la variabile di lock
- Se è a 0, la imposta ad 1, riabilita gli interrupt e usa la risorsa, altrimenti riabilita gli interrupt e si mette in attesa

La disabilitazione/riabilitazione degli interrupt rende la sequenza atomica, evitando una corsa critica se più processi tentassero di ottenere il lock contemporaneamente. Molti processori forniscono un supporto hardware, l'istruzione TEST-AND-SET, che esegue proprio questa sequenza di istruzioni in un'istruzione hardware, che per definizione è indivisibile.
Al rilascio, il lock viene reimpostato a 0.
Semafori binari e generalizzati:
Un semaforo è una struttura dati gestita dal SO che nel caso binario può assumere 2 valori: 1 (risorsa libera), 0 (risorsa occupata). Il sistema fornisce 2 istruzioni per utilizzare un semaforo:

- Acquire(S), usata per richiedere una risorsa: controlla il valore e se è 0 mette in attesa il processo, altrimenti lo imposta a 0 e il processo può usare la risorsa
- Release(S), usata per rilasciare la risorsa: reimposta il valore a 1

Il funzionamento è quindi simile a quello delle variabili di lock, ma sono gestiti dal SO.
I semafori generalizzati funzionano in modo analogo, e permettono di gestire istanze multiple delle risorse: possono assumere qualsiasi valore >=0, inizializzato al numero di istanze disponibili.
Le operazioni sono:

- Acquire(S), controlla il valore e se è 0 mette in attesa il processo, altrimenti sottrae 1 al valore e il processo può usare la risorsa
- Release(S), aumenta di 1 il valore

Un possibile problema riguarda il modo con cui un processo si mette in attesa: un ciclo infinito che

controlla il valore del semaforo (busy wait). Sebbene questo sia semplice da realizzare, in caso di attese lunghe si spreca il tempo del processore inutilmente. Si può modificare l'operazione acquire perchè inserisca i processi che devono attendere in una coda associata al semaforo (che può essere schedulata, ponendo attenzione alla starvation), e la release perché ne estragga uno dalla coda (se ve ne sono). In questo modo, non si elimina del tutto il problema del busy wait (l'accesso alla coda infatti va sincronizzato), ma lo si minimizza, poiché se anche bisogna attendere per inserire un processo in coda perché un altro processo sta venendo inserito, l'attesa durerà poco poiché si tratta di poche istruzioni. Inoltre con i semafori si possono verificare situazioni di attesa circolare (deadlock).
Il problema più grave però è che il loro corretto utilizzo spetta al programmatore, il quale può utilizzarli erroneamente o non utilizzarli affatto, e se ciò accade, ci possono essere violazioni di mutua esclusione, o si possono verificare fenomeni di starvation.
Monitor:
I monitor innalzano ulteriormente il livello di astrazione. Si tratta di un tipo di dato astratto, che, oltre a dei dati, possiede alcuni metodi che sono eseguiti in mutua esclusione. Questa però è gestita a livello di linguaggio di programmazione, in particolare dal compilatore che farà uso delle corrette chiamate di sistema, e quindi non spetta più al programmatore il loro corretto utilizzo.
Ovviamente ciò funziona se si utilizzano i monitor.
Un programmatore può creare un monitor adatto alle proprie esigenze definendo delle variabili di tipo condition, su cui un processo Q può aspettare il verificarsi di una condizione con l'operazione wait. In seguito un processo P potrà eseguire l'operazione signal su una condition, che risveglierà un processo Q dalla coda. A questo punto P può attendere finchè Q lascia il monitor o si mette in attesa di una condizione (signal and wait), oppure viceversa (signal and continue), oppure può lasciare immediatamente il monitor.