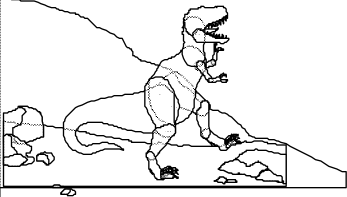
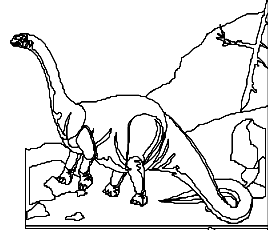
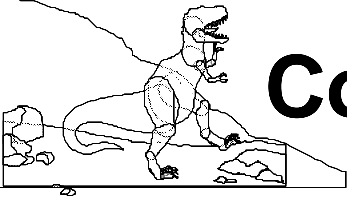


# Sincronizzazione fra processi

- Il problema della sezione critica
- Architetture di sincronizzazione



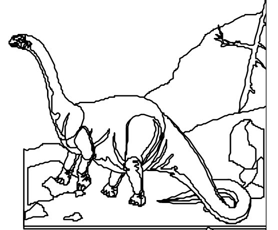


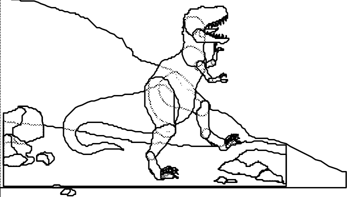


# Corse critiche

- *Corse Critiche* **(***Race condition***)**: situazioni nelle quali più processi accedono in concorrenza e modificano dati condivisi.
  - L’esito dell’esecuzione (il valore finale dei dati condivisi) dipende dall’ordine nel quale sono avvenute le operazioni.
- La lezione da imparare da quest’esempio è che è necessario proteggere le variabili condivise e che l’unico modo per farlo è controllare il codice che ha accesso alle variabili.
- Per evitare le corse critiche occorre che processi concorrenti vengano **sincronizzati**, in modo da impedire che più di un processo alla volta legga o scriva i dati condivisi.
- Nell’esempio precedente basta assicurare che un solo processo alla volta modifichi il valore della variabile **counter.**

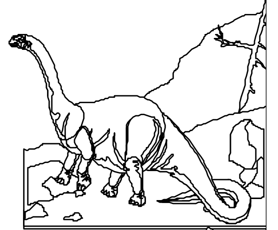


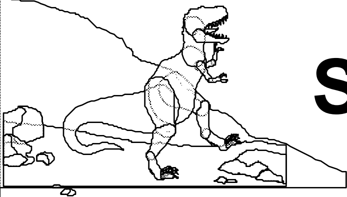




# Problema della sezione critica

- *n* processi che competono per utilizzare dati condivisi.
- Ciascun processo è costituito da un segmento di codice, chiamato *sezione critica*, in cui accede a dati condivisi.
- Problema: assicurarsi che, quando un processo esegue la sua sezione critica, a nessun altro processo sia concesso eseguire la propria.
  - ☞ L'esecuzione di sezioni critiche da parte di processi cooperanti è *mutuamente esclusiva* nel tempo.
- Soluzione: progettare un protocollo di cooperazione fra processi:
  - ☞ Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una *entry section*;
  - ☞ La sezione critica è seguita da una *exit section*; il rimanente codice è non critico.

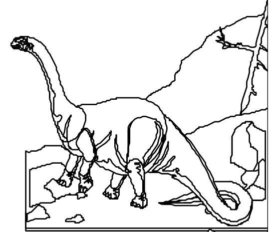


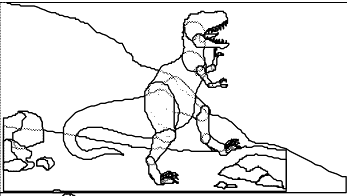


# Soluzioni al problema della sezione critica

Una soluzione al problema della sezione critica deve soddisfare questi 3 requisiti:

1. ***Mutua esclusione***. Se il processo $P_i$ è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. ***Progresso***. Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del prossimo processo che entrerà nella propria sezione critica dipende da quei processi che si trovano fuori dalle sezioni non critiche.
3. ***Attesa limitata***. Se un processo ha effettuato la richiesta di ingresso nella sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata.
   - Si assume che ciascun processo sia eseguito ad una velocità diversa da zero.
   - Non si fanno assunzioni sulla velocità relativa degli *n* processi.



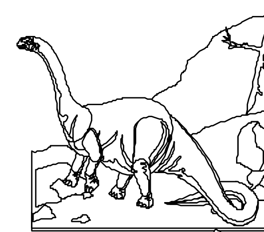


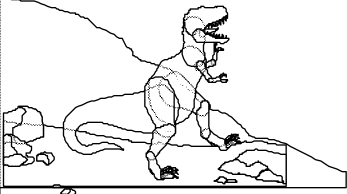
# Primi tentativi di soluzione del problema

- Solo 2 processi, $P_0$ e $P_1$.
- Struttura generale del processo $P_i$ (l’altro processo è $P_j$).

```
do {
    sezione d’ingresso
        sezione critica
    sezione d’uscita
        sezione non critica
} while (1);
```

- I processi possono condividere alcune variabili per sincronizzare le loro azioni.





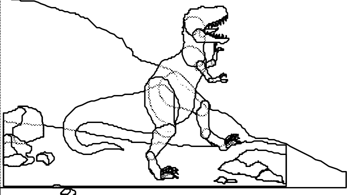
# Algoritmo 1

- Variabili condivise:
  - **int turn**;
    (inizialmente **turn = 0**).
  - **turn = i** ⇒ $P_i$ può entrare nella propria sezione critica.
- Processo $P_i$

```
do {
    while (turn != i);
        sezione critica
    turn = j;
        sezione non critica
} while (1);
```

*Richiede una stretta alternanza…*

- Soddisfa la **mutua esclusione**, ma non il **progresso**. Se *turn=0*, $P_1$ non può entrare nella propria sezione critica, anche se $P_0$ si trova nella propria sezione non critica.

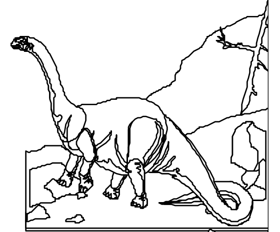


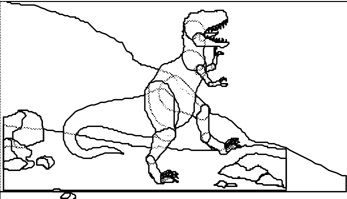
# Algoritmo 2

- Variabili condivise:
  - **boolean pronto [2]**;
    (inizialmente **pronto [0] = pronto [1] = false**).
  - **pronto [i] = true** ⇒ $P_i$ è pronto ad entrare nella propria sezione critica.
- Processo $P_i$

```
do {
    pronto [ i ] = true;
    while (pronto [ j ] );
        sezione critica
    pronto [ i ] = false;
        sezione non critica
} while (1);
```

- Soddisfa la **mutua esculsione**, ma non il **progresso**. I due processi possono porre entrambi *flag[i] = true*, bloccandosi indefinitamente.



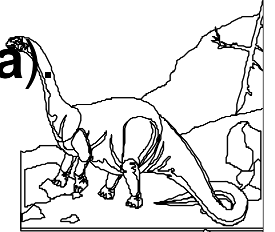


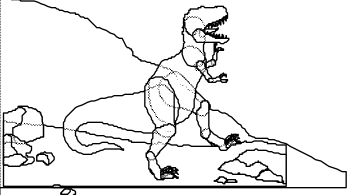
# Algoritmo 3 (Peterson)

- Combina le variabili condivise degli algoritmi 1 e 2.
- Processo $P_i$

```
do {
    pronto [ i ] = true;
    turn = j;
    while (pronto [ j ] && turn = j);
        sezione critica
    pronto [ i ] = false;
        sezione non critica
} while (1);
```

- Sono soddisfatte tutte le condizioni. Risolve il problema della sezione critica per due processi. $P_i$ entra nella sezione critica (**progresso**) al massimo dopo un'entrata di $P_j$ (**attesa limitata**).





# Algoritmo del fornaio

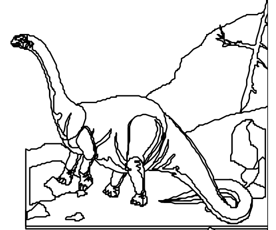
Soluzione del problema della sezione critica per *n* processi.

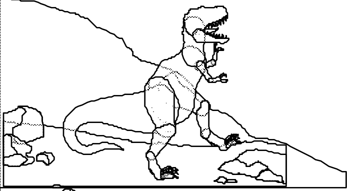
- Prima di entrare nella loro sezione critica, i processi ricevono un numero. Il possessore del numero più basso entra in sezione critica.
- Se i processi $P_i$ e $P_j$ ricevono lo stesso numero, se $i < j$, allora $P_i$ viene servito per primo, altrimenti tocca a $P_j$.
- Lo schema di numerazione genera sempre numeri non decrescenti; ad esempio, 1,2,3,3,3,3,4,5...
- Notazione <≡ ordine lessicografico (# biglietto, # processo)
  - ☞ $(a,b) < (c,d)$, se $a < c$ o se $a = c$ e $b < d$;
  - ☞ max $(a_0,\ldots, a_{n-1})$ è un numero $k$, tale che $k \geq a_i$ per $i = 0, \ldots, n - 1$.
- Variabili condivise:

  **boolean scelta [n];**

  **int numero [n];**

I vettori sono inizializzati rispettivamente a **false** e **0**.

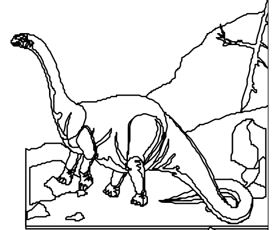


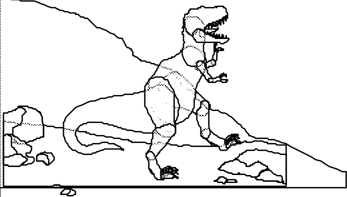


# Algoritmo del fornaio

```
do {
    scelta[ i ] = true;
    numero[ i ] = max(numero[ 0 ], numero[ 1 ], …, numero[ n – 1 ] ) + 1;
    scelta[ i ] = false;
    for ( j = 0; j < n; j++ ) {
            while (scelta[ j ]);
            while ( (numero[ j ] != 0) && (numero[ j ], j ) < (numero[ i ],i ));
    }
      sezione critica
    numero[ i ] = 0;
      sezione non critica
} while (1);
```

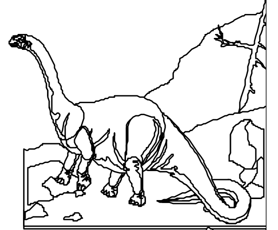
*number[i] = 0* indica che $P_i$ non vuole entrare in sezione critica.

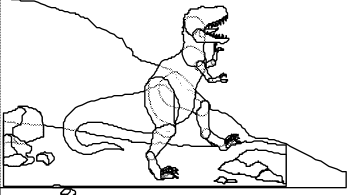




# Hardware di sincronizzazione

- In sistemi dotati di una singola CPU due processi concorrenti non possono essere eseguiti contemporaneamente.
- Per garantire la mutua esclusione è sufficiente evitare che un processo venga interrotto.
- Questa funzionalità può essere fornita come primitive del kernel per abilitare e disabilitare le interruzioni.
- Tale scelta può portare ad un degrado delle prestazioni
  - Impossibilità da parte del processore di interfogliare le operazioni;
  - In sistemi multiprocessori, necessità di trasmettere la richiesta di disabilitazione a tutte le CPU.
- Molte architetture forniscono particolari istruzioni che permettono di controllare e modificare il contenuto di due parole di memoria in maniera **atomica.**





# TestAndSet

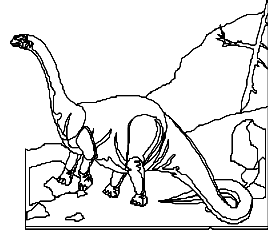
- Controlla e modifica il contenuto di una parola di memoria:

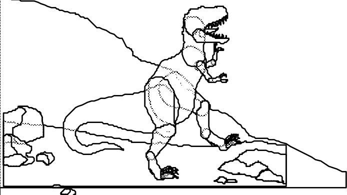
.

```
boolean TestAndSet ( boolean &obiettivo ) {
   boolean valore = obiettivo;
   obiettivo = true;

   return valore;
}
```

- L’intera operazione è eseguita in maniera atomica.





# Mutua esclusione con TestAndSet

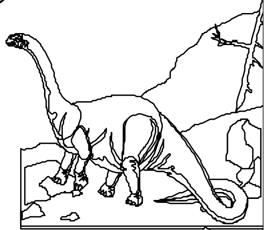
- Dati condivisi:

```
boolean blocco = false;
```

- Processo $P_i$

```
do {
    while ( TestAndSet ( blocco ) ) ;
        sezione critica
    blocco = false;
        sezione non critica
}while(1);
```

Non soddisfa l'attesa limitata

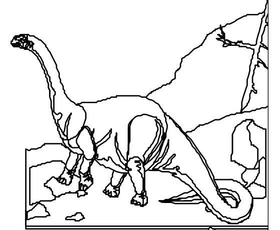


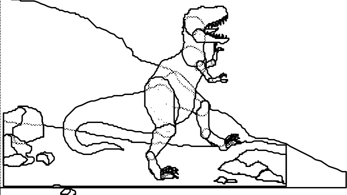
# Swap

- Scambia il contenuto di due parole di memoria:

```
void Swap ( boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

- L’intera operazione è eseguita in maniera atomica.





# Mutua esclusione con Swap

- Dati condivisi:

  **boolean blocco = false;**

- Processo $P_i$

```
do {
        chiave = true;
        while ( chiave == true )
                Swap (blocco, chiave);
        sezione critica
    blocco = false;
        sezione non critica
}while(1);
```

**Non soddisfa l'attesa limitata**