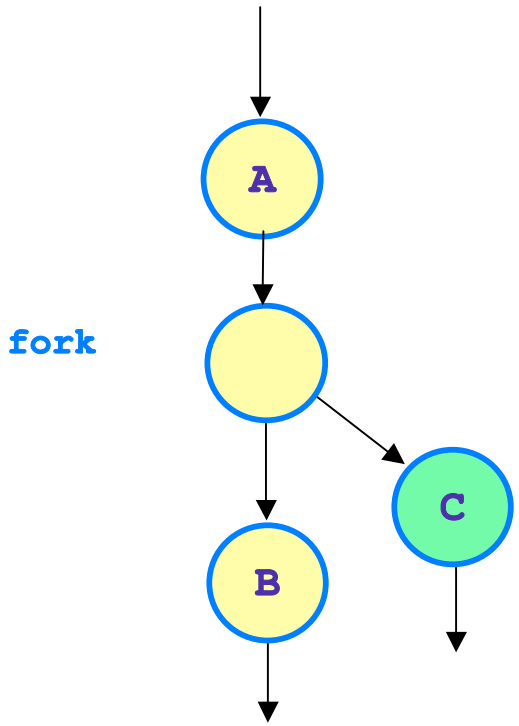
Costrutti linguistici per la specifica della concorrenza

Fork/Join

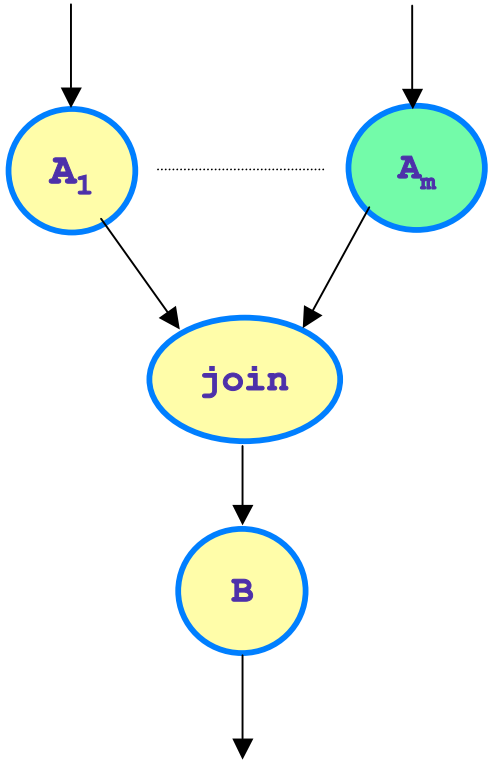
L'esecuzione di una fork coincide con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante

```
  /* processo p: */
    =======
A:  ........;
    p = fork fun;
B:  ........;
    =======
    =======

/* codice nuovo processo:*/
void  fun()
{
    C: ........;
    =======
}
```

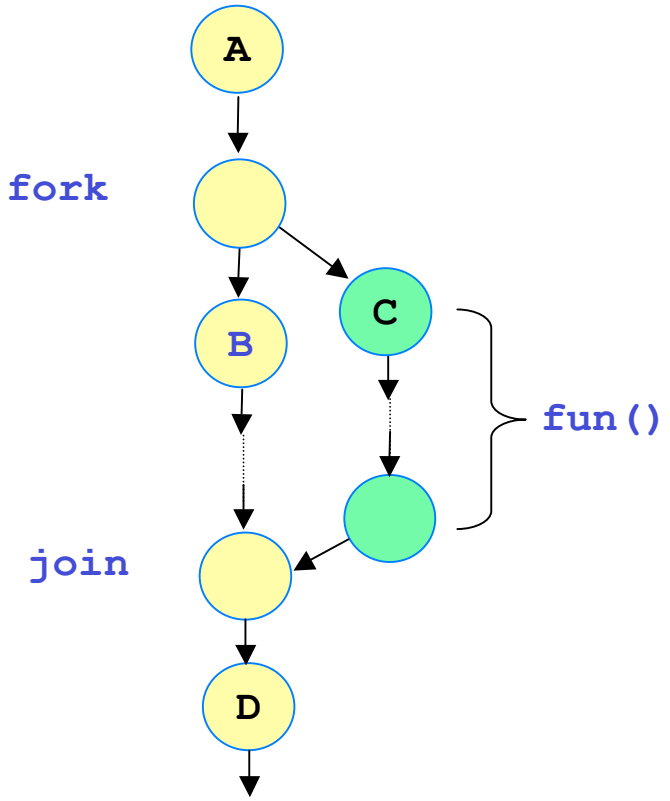


La join consente di determinare quando un processo , creato tramite la fork, ha terminato il suo compito, sincronizzandosi con tale evento.



```
  /* processo p: */
     ...
A:   ........;
     p = fork fun;
B:   ........;
     join p;
D:   ........;
     ...

/* codice nuovo processo:*/
void  fun()
{
     C: ........;
     ...
}
```

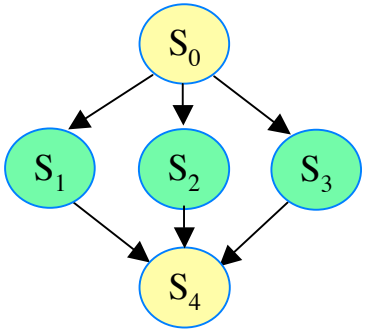


Possibilità di denotare in modo esplicito il processo sulla cui terminazione ci si vuole sincronizzare.

Cobegin-Coend

La concorrenza viene espressa nel modo seguente:

```
S0;
cobegin
   S1;
   S2;
   S3;
coend
S4;
```



Le istruzioni S1, S2,…,Sn sono eseguite in parallelo. Ogni S_i può contenere altre istruzioni cobegin..coend al suo interno.

Processo

Costrutto linguistico per individuare, in modo sintatticamente preciso, quali moduli di un programma possono essere eseguiti come processi autonomi:

```
process <identificatore>(<parametri formali>)
 {  <dichiarazione di variabili locali>;
    <corpo del processo>
 }
```

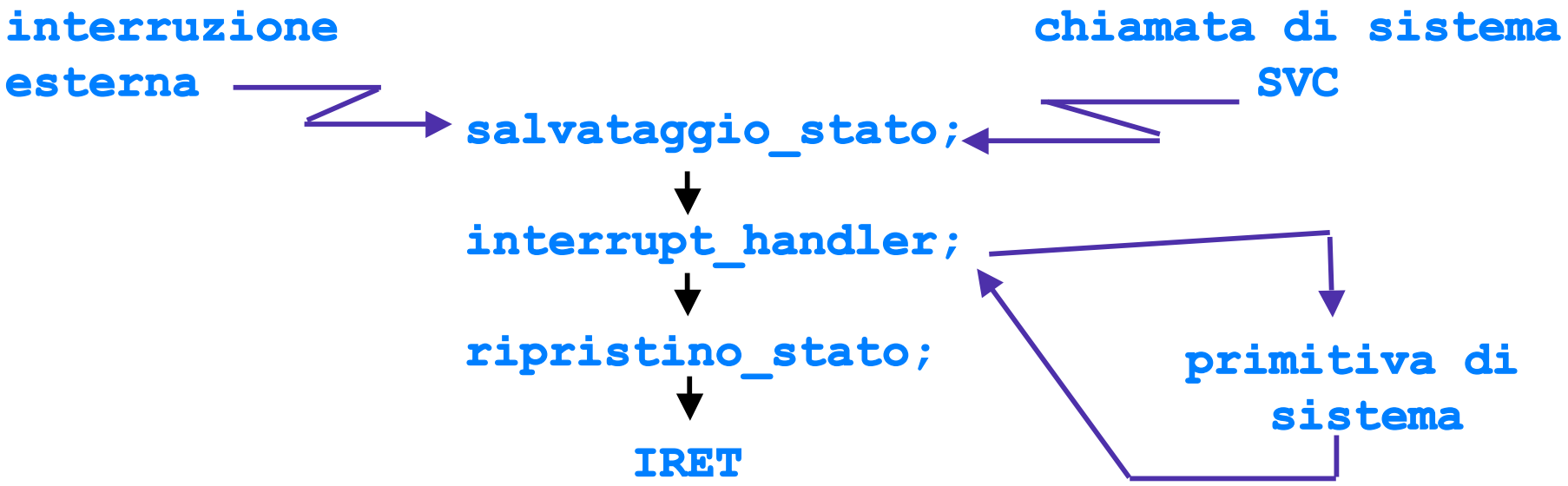
Il Nucleo di un sistema monoprocessore: realizzazione delle *primitive* di specifica della concorrenza

fork usata da un processo per creare un altro processo.

join usata per consentire ad un processo di attendere la terminazione di un processo *figlio* (equivalente alla **wait** di Unix).

quit usata per eliminare un processo (equivalente alla **exit** di Unix).

Passaggio del controllo fra ambiente dei processi e ambiente di nucleo



Rappresentazione dei processi: descrittori di processo

```
typedef des_processo * p_des;


typedef struct {
   PID nome;
  modalità_di_servizio servizio;
  tipo_contesto  contesto;
  tipo_stato  stato;
  PID  padre;
  int  N_figli;
  des_figlio prole[max_figli];
  p_des successivo;
}des_processo;
```

Ogni processo viene identificato univocamente con un numero intero:

```
typedef int PID;
```

Funzione (assegna_nome) che, chiamata all'atto della creazione di un processo, restituisce il suo PID:

```
PID assegna_nome();
```

Modalità di servizio (es.due campi):

```
typedef struct {
        int priorità;
        int delta_t;
} modalità_di_servizio;
```

Descrittore del figlio (des_figlio):

```
typedef struct{
      PID figlio;
      boolean terminato; <true = terminato>
} des_figlio;
```

Insieme di tutti i descrittori (descrittori):

```
des_processo descrittori[num_max_proc];
```

Funzione per scegliere un descrittore (descrittore):

```
p_des descrittore(PID x);
```

Descrittore coda (des_coda):

```
typedef struct {
          p_des  primo, ultimo;
          }des_coda;
```

Inserimento e prelievo in/da coda:

```
     void Inserimento (p_des pro,des_coda coda);
     p_des Prelievo (des_coda coda);
```

Code dei processi pronti:

```
des_coda coda_processi_pronti[num_min_priorità];
```

Coda dei descrittori liberi:

```
          des_coda descrittori_liberi;
```

Cambio di contesto

```
p_des  processo_in_esecuzione;

void salvataggio_stato () {
      p_des esec = processo_in_esecuzione;
      esec -> contesto = <valori dei registri della CPU>;
}

void ripristino_stato( ) {
      p_des esec = processo_in_esecuzione;
      <registri della CPU>= esec -> contesto;
      }
```

Schedulazione

```
void assegnazione_CPU {
  int  k=0;
  p_des p;
  while (coda_processi_pronti[k].primo)==null)
         k++;
  p = prelievo(coda_processi_pronti [k]);
  processo_in_esecuzione = p;
  <registro-temporizzatore>= p -> servizio.delta_t;
}
```

Attivazione di un processo

```
void attiva (p_des proc) {
  p_des esec = processo_in_esecuzione;
  int pri_esec = esec -> servizio.priorità;
  int pri_proc = proc -> servizio.priorità;
  proc -> stato = <"processo attivo">;
  if (pri_esec > pri_pro) { /* pre-emption*/

inserimento(esec,coda_processi_pronti[pri_esec]);
    processo_in_esecuzione = proc;
    }
  else
    inserimento(proc,coda_processi_pronti[pri_proc]);
}
```

fork

```
typedef enum {OK,eccezione} risultato;


risultato fork (des_processo inizializzazione) {
    p_des  p;
    int NF;
    p_des esec = processo_in_esecuzione;
    if (descrittori_liberi.primo == NULL )
        return eccezione; /* non ci sono descritt. liberi*/
    else {
        p = prelievo(descrittori_liberi);
        *p = inizializzazione;
        p -> nome = assegna_nome();
        p -> padre = esec -> nome;
        NF = esec->N_figli;
        esec -> prole[NF].figlio = p -> nome;
        esec -> prole[NF].terminato = false;
        esec -> N_figli++;
        attiva(p);
        return ok; }
}
```

join

```
void join (PID nome_figlio) {
    p_des esec = processo_in_esecuzione;
    int  k =indice_figlio(esec,nome_figlio);
    if (esec -> prole[k].terminato == false)
    { /* figlio non terminato*/
        esec -> stato = <“sospeso in attesa che il figlio termini”>;
        Assegnazione_ CPU();
    }
}
```

quit

```
void  quit(){
  p_des esec = processo_in_esecuzione;
  PID  nome_padre = esec -> padre;
  p_des p_padre = descrittore(nome_padre);
  int k=indice_figlio(p_padre, esec -> nome);
  p_padre -> prole[k].terminato=true;
  inserimento(esec, descrittori_liberi);
  if (p_padre -> stato==<“ in attesa che questo figlio termini”>)
  {
     int pri = p_padre -> servizio.priorità;
     inserimento(p_padre,coda_processi_pronti[pri]);
     p_padre -> stato = <“processo attivo”>;
  }
  assegnazione_CPU ( );
}
```

Time sharing

• Per consentire la realizzazione di modalità di servizio a *divisione di tempo* occorre revocare ad intervalli fissati di tempo la CPU al processo in esecuzione e di assegnarla ad un nuovo processo pronto:

```
void interruzione_temporizzatore () {
      p_des esec=processo_in_esecuzione;
      int k;
      k=esec->servizio.priorità;
      Inserimento(esec,coda_processi_pronti[k]);
      assegnazione_CPU();
}
```