Lezione 4:

I PROCESSI

- Concetto di processo
- Realizzazione dei processi
- Scheduling dei processi
- Operazioni sui processi
- Processi cooperanti
- Comunicazione fra processi

Cosa e' un processo?

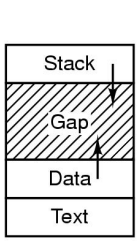
- Un sistema operativo esegue programmi di varia natura:
  - Compilatori, word processor, programmi utente, programmi di sistema,…

Processo = un programma in esecuzione;

- l'esecuzione di un processo avviene in modo sequenziale.
- Ad un programma possono corrispondere piu' processi
- Termini sinonimi: task, job

processi

- Un processo e' una entita' dinamica (il suo stato varia nel tempo)
- Ad ogni processo il sistema operativo assegna un'area di memoria

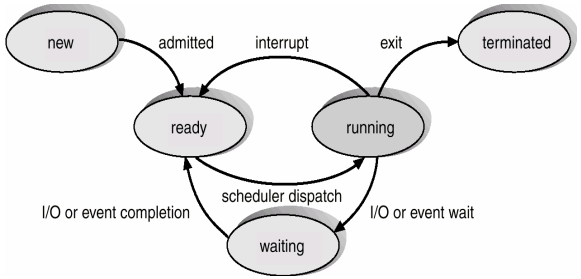


- area testo: contiene il codice eseguibile
- area dati: contiene le variabili globali
- area stack: contiene le variabili locali, e gli indirizzi di rientro delle subroutine

Stato del processo

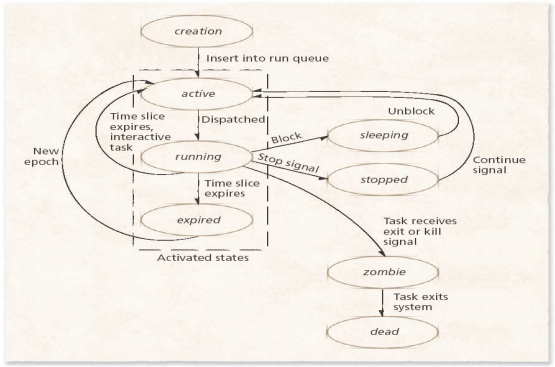
- In un sistema multiprogrammato, mentre viene eseguito un processo cambia **stato**:
  - **New** (nuovo): Il processo viene creato.
  - **Running** (in esecuzione): Le istruzioni vengono eseguite.
  - **Waiting** (in attesa): Il processo è in attesa di un evento.
  - **Ready** (pronto): Il processo è in attesa di essere assegnato ad un processore.
  - **Terminated** (terminato): Il processo ha terminato la propria esecuzione.

Diagramma degli stati fondamentali di un processo



- sempre un solo processo running
- molti processi ready o waiting
- la cpu e' utilizzata a turno dai vari processi (context switch)

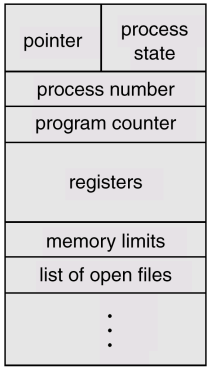
Esempio: Linux



L'effettiva denominazione degli stati dipende dal sistema

Come il S.O. riconosce e gestisce i processi?

Per poter effettuare correttamente il context switch, il sistema operativo deve salvare una serie di informazioni in un opportuna struttura dati conservata nel kernel: Process Control Block



- Stato del processo
- process id
- Program counter
- Registri della CPU
  (accumulatori, indice, stack pointer)
- Informazioni sullo scheduling della CPU
  (priorità, puntatori alle code di scheduling)
- Informazioni sulla gestione della memoria
  (registri base e limite, tabella pagine/segmenti)
- Informazioni di contabilizzazione delle risorse
  (numero job/account, tempo di CPU)
- Informazioni sullo stato di I/O
  (lista dispositivi/file aperti)

Campi del Process Control Block

La struttuta del PCB dipende dall'architettura e dal S.O.

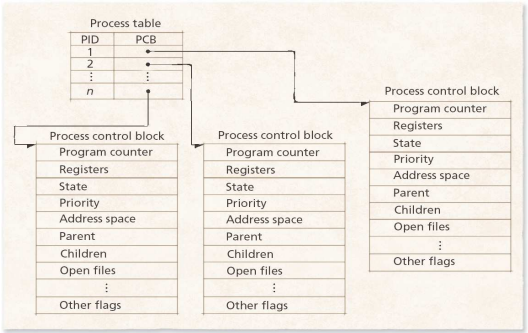
Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Struttura non unica !!!

Process table

Il sistema operativo mantiene un puntatore ad ogni PCB in opportune Process table (per utente o generale)

Quando un processo e' terminato, il sistema operativo rimuove il processo dalla Process Table e libera le risorse



Code per lo scheduling di processi

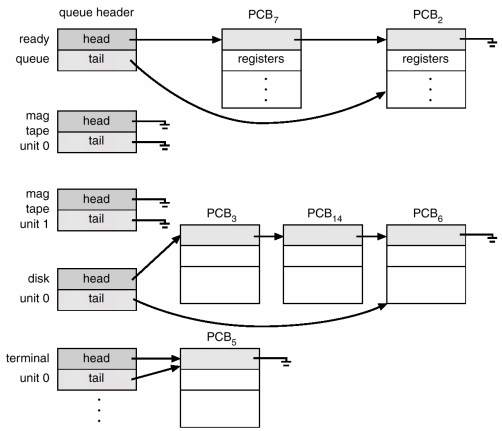
In un sistema operativo (monoprocessore) c'e' sempre un solo processo in esecuzione

Gli altri processi sono conservati in opportune code

- **Ready queue** (Coda dei processi pronti) — Insieme di tutti i processi pronti ed in attesa di esecuzione, che risiedono i memoria centrale.
- **I/O queue** (Coda dei dispositivi) — Insieme di processi in attesa per un dispositivo di I/O.

Le PCB dei processi si "spostano" fra le varie code.

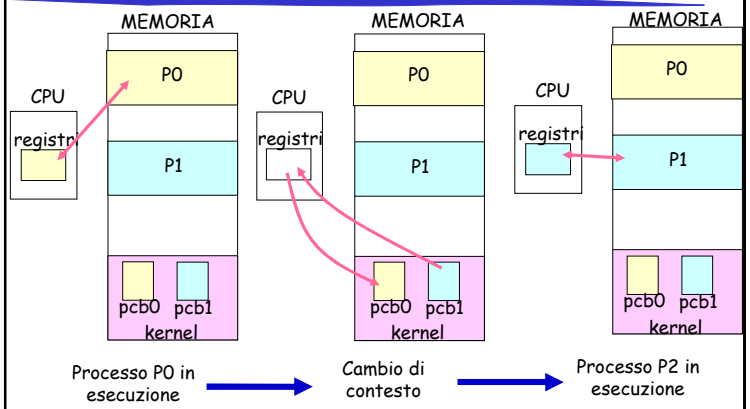
Ready queue e code ai dispositivi di I/O



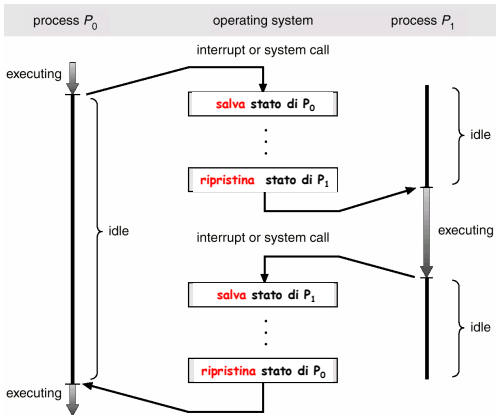
Context Switch

- Quando la CPU passa da un processo all'altro, il sistema deve salvare lo stato del vecchio processo e caricare lo stato precedentemente salvato per il nuovo processo.
- il sistema non lavora utilmente mentre cambia contesto.
- Il tempo di context-switch e' un sovraccarico per il sistema e dipende dal supporto hardware (velocità di accesso alla memoria, numero di registri da copiare, istruzioni speciali, gruppi di registri multipli).

Context switch



Context switch



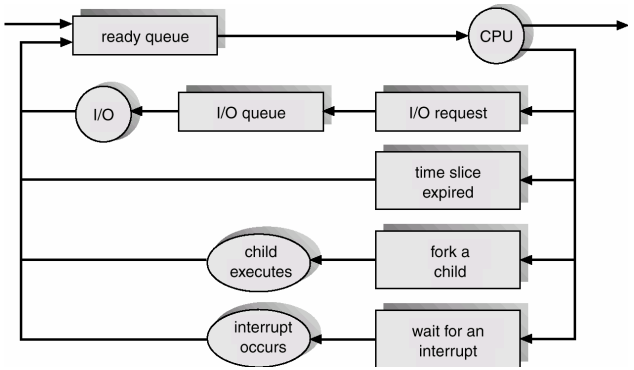
Chi decide quale processo deve usare la cpu?

Un opportuno programma decide quale
tra i processi nella ready queue deve utilizzare la CPU

SCHEDULER

- Scheduler a **breve termine** (o scheduler della CPU) :
  - seleziona quale processo debba essere eseguito successivamente, ed alloca la CPU.
  - Deve essere chiamato molto spesso (~ 100 msec)
  - Deve essere veloce (~ 1 msec)
  - E' responsabile dei tempi di attesa

Scheduler a breve termine

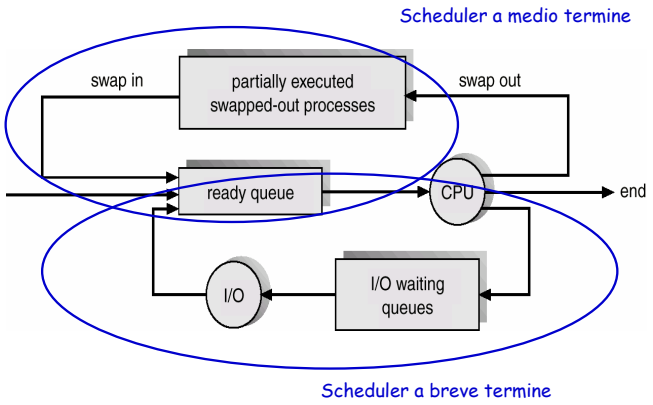


Tipi di scheduler

- Scheduler a **lungo termine** (o scheduler dei job):
  - seleziona quali processi devono essere portati dalla memoria di massa in memoria centrale.
  - Controlla il numero di processi in un sistema
  - Puo' essere chiamato ogni volta che si crea un nuovo processo (secondi o minuti)
  - Puo' essere assente nei moderni S.O. (UNIX)

- Scheduler a **medio termine** (*swapper*):
  - rimuove processi dalla memoria (e dalla contesa per la CPU) e riduce il grado di multiprogrammazione.
  - E' responsabile del numero di processi presenti in memoria e facilita il compito dello scheduler a breve termine
  - Assieme allo scheduler a lungo termine determina la composizione delle code dei processi

Interazione scheduler a breve e medio termine



scheduler a medio/lungo termine

I processi possono essere classificati in:

- *Processi I/O-bound*: impiegano più tempo effettuando I/O rispetto al tempo impiegato per elaborazioni
  - in generale, si hanno molti *burst* di CPU di breve durata.
  - Utilizzano prevalentemente le code dei dispositivi

- *Processi CPU-bound*: impiegano più tempo effettuando elaborazioni
  - in generale, si hanno pochi *burst* di CPU di lunga durata.
  - utilizzano prevalentemente la ready queue

OBIETTIVO

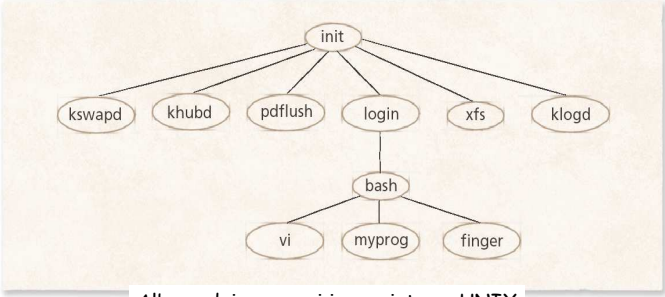
Avere sempre processi presenti nelle code dei processi pronti e dei dispositivi

Operazioni sui processi

- Tutti i sistemi operativi forniscono i servizi fondamentali per la gestione dei processi, tra cui:
  - Creazione processi
  - Distruzione processi
  - Sospensione processi (rimozione dalla memoria)
  - Ripresa processi
  - Cambio priorita' dei processi
  - Blocco processi (rimozione dalla cpu)
  - Sblocco processi
  - Attivazione processi
  - Comunicazione tra processi

Come vengono creati i processi?

un processo (padre) puo' creare numerosi processsi (figli), che, a loro volta, possono creare altri processi, formando un albero (genealogico) di processi.



Albero dei processi in un sistema UNIX

Creazione dei processi: diverse scelte progettuali

Risorse :

- Il padre e il figlio condividono tutte le risorse.
- I figli condividono un sottoinsieme delle risorse del padre.
- Il padre e il figlio non condividono risorse.

Minor carico nel sistema

↕

Maggior carico nel sistema

Spazio degli indirizzi

- Il processo figlio è un duplicato del processo padre (UNIX).
- Nel processo figlio è caricato subito un diverso programma (VMS).

Esecuzione

- Il padre e i figli vengono eseguiti concorrentemente.
- Il padre attende la terminazione dei processi figli.

Un processo termina quando

- esegue l'ultima istruzione e chiede al sistema operativo di essere cancellato per mezzo di una specifica chiamata di sistema (**exit** in UNIX)
  - Può restituire dati al processo padre
  - Le risorse del processo vengono deallocate dal SO.
- Viene terminato dal padre quando, ad esempio:
  - Il figlio ha ecceduto nell'uso di alcune risorse.
  - Il padre termina (in alcuni sistemi)
    - *terminazione a cascata*
- Viene terminato da un altro processo (eventualmente il padre) per mezzo di una specifica chiamata di sistema (**abort** in UNIX)

Esempio: UNIX

- la funzione fork crea un nuovo processo
  - il figlio viene creato copiando tutto il PCB del padre
  - ritorna 0 (zero) nel figlio
  - ritorna il pid del figlio nel padre
- la funzione execlp carica nel nuovo processo un programma
  - vengono sostituite le aree testo, data e stack
- la funzione exit fa terminare un processo
  - eventualmente comunica al padre lo stato di uscita
- la funzione wait fa attendere al padre la terminazione del figlio
  - eventualmente riceve dal figlio lo stato di uscita

Esempio: UNIX

```
int main ()  {
     int pid;
     …
     pid = fork( );
     if( pid == 0 )    {
          execlp("/bin/ls","ls", NULL);

     }else{
          wait(NULL);

     }
     …
     exit(0);
}
```

Processo figlio

Processo padre

A chi viene restituito 0
(chi e' il padre del main) ?

Esempio: una semplice shell di Unix

```
while (TRUE) {                          /* repeat forever */
     type_prompt( );                    /* display prompt on the screen */
     read_command(command, params);     /* read input line from keyboard */

     pid = fork( );                     /* fork off a child process */
     if (pid < 0) {
          printf("Unable to fork0);     /* error condition */
          continue;                     /* repeat the loop */
     }

     if (pid != 0) {
          waitpid (-1, &status, 0);     /* parent waits for child */
     } else {
          execve(command, params, 0);   /* child does the work */
     }
}
```


Processi cooperanti

- Un processo è **indipendente** se non può influire su altri processi nel sistema o subirne l'influsso.
- Processi **cooperanti** possono influire su altri processi o esserne influenzati.
- La presenza o meno di dati condivisi determina univocamente la natura del processo.
- Vantaggi della cooperazione fra processi
  - Condivisione di informazioni
  - Accelerazione del calcolo (in sistemi multiprocessore)
  - Modularità
  - Convenienza

Comunicazione tra processi (IPC)

Un sistema operativo e' composto da numerosi moduli che devono interagire tra loro

Necessita' di un meccanismo di comunicazione tra processi

- segnali
- Memoria condivisa
- scambio di messaggi

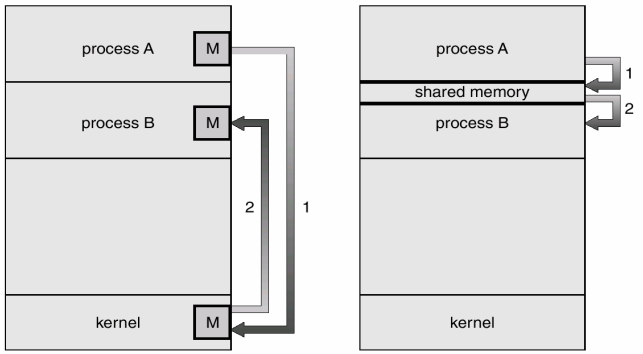
segnali

- Interruzioni software per la comunicazione asincrona tra processi
  - Non permettono ai processi di scambiarsi dati
  - Il processo che riceve il segnale non e' in un particolare stato di attesa (evento asincrono)
  - I processi possono catturare, ignorare o mascherare un segnale
    - Catturare un segnale significa far eseguire al sistema operativo una specifica routine al momento della ricezione del segnale o una azione di default associata al segnale
    - Ignorare un segnale significa far eseguire al sistema operativo delle operazioni di dafult associate al segnale
    - Mascherare un segnale significa istruire il sistema operativo a non consegnare il segnale fino a nuovo ordine

Principali segnali di Unix

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Scambio di messaggi / memoria condivisa



Mediante scambio di messaggi

Mediante memoria condivisa

Scambio di messaggi

- Lo scambio di messaggi consente due operazioni:
  - send(*messaggio*) — la dimensione del messaggio può essere fissa o variabile;
  - receive(*messaggio*).
- Se i processi *P* e *Q* vogliono comunicare, devono:
  - stabilire fra loro un canale di comunicazione;
  - scambiare messaggi per mezzo di send/receive.
- Implementazione del canale di comunicazione:
  - fisica (es. memoria condivisa, bus hardware);
  - logica (proprietà logiche).

Problemi di implementazione

- Come vengono stabiliti i canali (connessioni)?
- È possibile assegnare un canale a più di due processi?
- Quanti canali possono essere stabiliti fra ciascuna coppia di processi comunicanti?
- Qual è la capacità di un canale?
- Il formato del messaggio che un canale può gestire è fisso o variabile?
- Stabilire canali monodirezionali o bidirezionali?

Comunicazione diretta o simmetrica

- I processi devono dichiarare esplicitamente i loro interlocutori:
  - **send** (*P, messaggio*) — invia un messaggio al processo *P*
  - **receive**(*Q, messaggio*) — riceve un messaggio dal processo *Q*
- Proprietà del canale di comunicazione:
  - I canali vengono stabiliti automaticamente la momento della send/receive
  - Ciascun canale è associato esattamente ad una coppia di processi.
  - Tra ogni coppia di processi comunicanti esiste esattamente un canale.
  - Il canale può essere unidirezionale (ogni processo collegato al canale può soltanto trasmettere/ricevere), ma è normalmente bidirezionale.

Comunicazione indiretta o asimmetrica

- I messaggi vengono inviati/ricevuti da ***mailbox*** (*porte*).
  - Ciascuna mailbox è idendificata con un ***id*** unico.
  - I processi possono comunicare solamente se condividono una mailbox.
- Proprietà dei canali di comunicazione:
  - Un canale viene stabilito solo se i processi hanno una mailbox in comune.
  - Un canale può essere associato a più processi.
  - Ogni coppia di processi può condividere più canali di comunicazione.
  - I canali possono essere unidirezionali o bidirezionali.
- Operazioni:
  - creare una nuova mailbox
  - Inviare/ricevere messaggi attraverso mailbox
  - distruggere una mailbox

Comunicazione indiretta

- Primitive di comunicazione:
  - **send**(*A, messaggio*) — invia un messaggio alla mailbox *A*
  - **receive**(*A, messaggio*) — riceve un messaggio dalla mailbox *A*
- Problema nella condivisione di mailbox
  - P_1, P_2, e P_3 condividono la mailbox *A*.
  - P_1, invia; P_2 e P_3 ricevono.
  - Chi riceve il messaggio?
- Soluzioni:
  - Permettere ad un canale di essere associato ad al più due processi.
  - Permettere ad un solo processo alla volta di eseguire un'operazione di ricezione.
  - Permettere al SO di selezionare arbitrariamente il ricevente. Il sistema comunica l'identità del ricevente al trasmittente.

Sincronizzazione

Le primitive **send** e **receive** possono essere sia bloccanti (o sincrone) che non-bloccanti (o asincrone)

- Invio bloccante: il processo che invia attende che il processo ricevente riceva il messaggio
- Invio non bloccante: il processo che invia riprende subito l'elaborazione
- Ricezione bloccante: il processo ricevente attende l'arrivo del messaggio
- Ricezione non bloccante: il processo ricevente riceve un messaggio valido oppure nullo

Bloccante = sicuro Non bloccante = efficiente

Implementazione

- La coda dei messaggi legata ad un canale può essere implementata in tre modi.

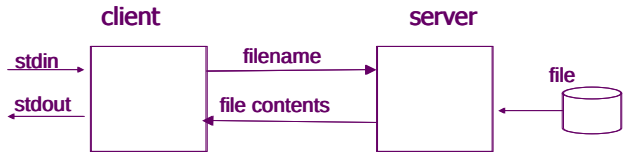
1. **Capacità zero** — Il canale non può avere messaggi in attesa al suo interno. Il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio (*rendezvous*).
2. **Capacità limitata** — Lunghezza finita pari a *n* messaggi. Se il canale è pieno, il trasmittente deve attendere.
3. **Capacità illimitata** — Lunghezza infinita. Il trasmittente non attende mai.

Condizioni di eccezione

- **Terminazione del processo**: un processo trasmittente o ricevente può terminare prima che un messaggio sia stato elaborato ⇒ messaggi mai ricevuti, processi bloccati in attesa.
- **Messaggi perduti**: a causa di guasti sui canali di comunicazione; il SO o il processo trasmittente sono responsabili del rilevamento della condizione di eccezione e della ripetizione del messaggio.
- **Messaggi alterati**: a causa di disturbi sul canale di comunicazione; il messaggio deve essere ritrasmesso.

Esempio: client server

- E' un classico paradigma di comunicazione tra processi
- un processo P (il client) chiede un servizio ad un processo Q (il server).
- Es: apertura di un file

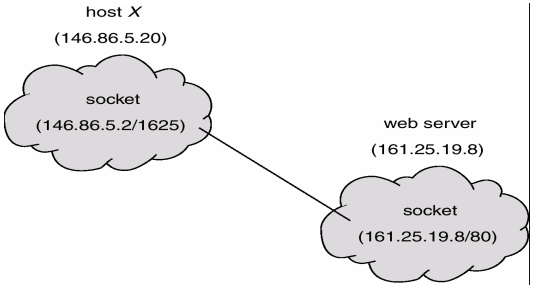


Scambio di messaggi in sistemi distribuiti

- I messaggi trasmessi si possono perdere
  - Necessita' di protocolli che confermino la corretta ricezione del messaggio attraverso "ricevute di ritorno" (es. TCP/IP)
  - Meccanismi di timeout per ritrasmettere i messaggi se la ricevuta non ritorna
- Processi su macchine differenti possono avere lo stesso id e possono portare a trasmissioni incorrette
  - I messaggi sono passati tra computer usando indirizzi e porte su cui i processi sono posti in ascolto
- Problemi chiave
  - Sicurezza e autenticazione

Esempio: i socket

- Un socket e' il terminale di un canale di comunicazione.
- E' la concatenazione di un indirizzo IP e una porta
- Il socket **161.25.19.8:80** si riferisce alla porta **80** sull' host **161.25.19.8**



Esempio: problema del produttore-consumatore

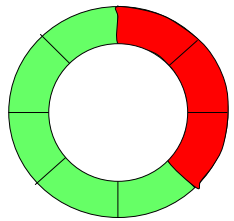
- È un paradigma classico per processi cooperanti. Il processo **produttore** produce informazioni che vengono consumate da un processo **consumatore**.
  - *Buffer illimitato:* non vengono posti limiti pratici alla dimensione del buffer.
  - *Buffer limitato:* si assume che la dimensione del buffer sia fissata.
- Esempio: Un programma di stampa produce caratteri che verranno consumati dal driver della stampante.

Soluzione con buffer limitato e memoria condivisa

- Dati condivisi

```
#define BUFFER_SIZE 8
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Es: BUFFER_SIZE = 8
in = 3 out = 0



- in indice della prima posizione libera
- out indice della prima posizione occupata

Soluzione con buffer limitato e memoria condivisa

```
item nextProduced;
while (1) {
        while ( ((in + 1) % BUFFER_SIZE) == out ) ;
                /* buffer pieno: non fare niente */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Processo Produttore

```
item nextConsumed;
while (1) {
        while (in == out) ;
                /* buffer vuoto: non fare niente  */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Processo Consumatore

osservazione

La soluzione proposta consente l'utilizzo di soli BUFFER_SIZE-1 elementi

PERCHE'?

Trovare una soluzione che utilizza tutti i BUFFER_SIZE elementi