Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2021-2022

Pietro Frasca

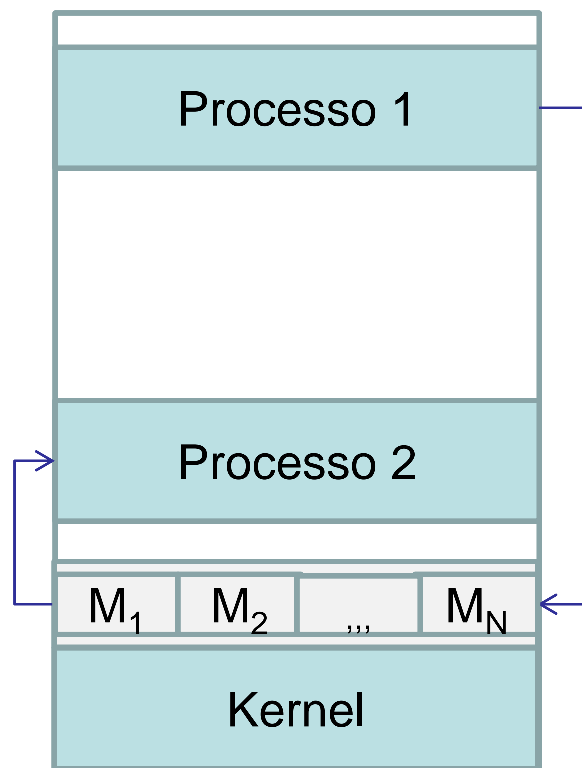
Lezione 7

Martedì 2-11-2021

Scambio di messaggi (message passing)

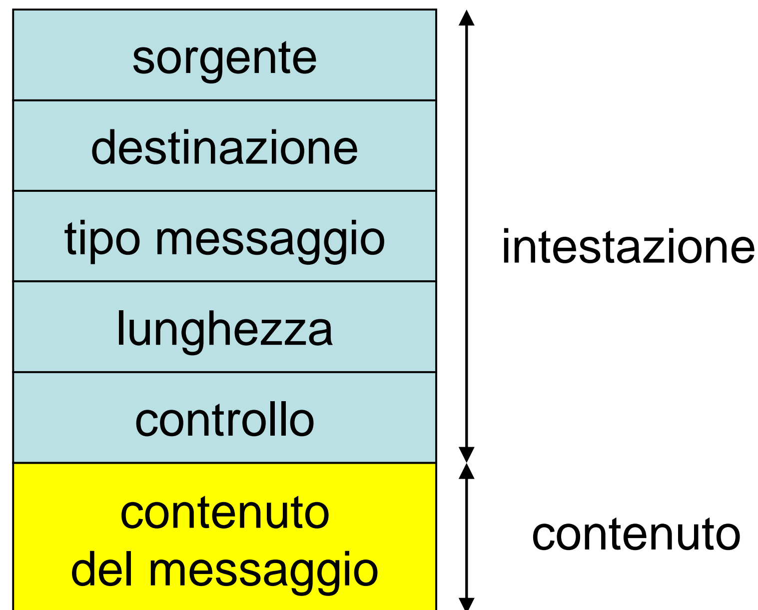
- Questa tecnica consente ai processi di comunicare e di sincronizzare le loro attività senza condividere una stessa area di memoria comune.
- È particolarmente usata in un ambiente distribuito, in cui i processi comunicanti sono eseguiti su diversi computer collegati in rete.
- Tuttavia il *message passing* è usato anche in sistemi a memoria condivisa, sia a singolo processore che multiprocessore.
- In quest'ultimo caso il canale di comunicazione è costituto da un'area di memoria, gestita dal sistema operativo, nella quale vengono scritti e letti i messaggi.
- Una tecnica a scambio di messaggi deve fornire almeno due funzioni di sistema basilari. La prima per inviare un messaggio e l'altra per riceverlo. Indicheremo queste due funzioni con ***send(message)***, e ***receive(message)***.
- Se più processi vogliono comunicare è necessario che esista un canale di comunicazione tra di essi. Questo canale può essere realizzato sia a livello hardware che software in vari modi.

- A livello logico, ci sono diversi metodi per implementare un canale di comunicazione e le funzioni *send()* e *receive():*



Modello a scambio di messaggi

- Un messaggio ha una struttura ben precisa. Tipicamente è composto da due parti, l'***intestazione*** e il **contenuto (payload)**. Campi tipici dell'intestazione sono l'identificatore del mittente e del destinatario, il tipo di messaggio, la lunghezza del messaggio, ecc. Il contenuto include il messaggio vero e proprio.



Comunicazione diretta e indiretta

- I processi per comunicare devono potersi identificare a vicenda. Essi possono utilizzare la comunicazione ***diretta*** o ***indiretta***.
- Con la ***comunicazione diretta***, ogni processo che vuole comunicare deve identificare esplicitamente l'altra estremità della comunicazione.
- In questo schema, le *send()* e *receive()* sono definite con le seguenti firme:

send (destinatario, messaggio) // invia un messaggio al processo destinatario.

receive (mittente, messaggio) // riceve un messaggio dal processo mittente.

- Questo modello di comunicazione ha una forma simmetrica d'indirizzamento; cioè, sia il processo mittente che il processo ricevente devono specificare l'identità dell'altro per comunicare.

La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione diretta.

PRODUTTORE

```
pid cons = …
main(){
  Messaggio mes;
  do {
    produci (mes);
    send (cons, mes);
  while (!fine)
}
```

CONSUMATORE

```
pid prod =…
main(){
  Messaggio mes;
  do {
    receive (prod, mes);
    consuma (mes)
  while (!fine)
}
```

Schema di comunicazione diretta simmetrica

- La ***comunicazione indiretta*** è una variante di questo schema che usa un indirizzamento asimmetrico piuttosto che simmetrico.
- Con questa tecnica, solo il mittente deve indirizzare il destinatario; il destinatario non è tenuto a conoscere il nome del mittente.
- In questo schema, le funzioni *send()* e *receive()* sono definite come segue:

send (destinatario, messaggio) *// invia un messaggio al processo destinatario.*

receive (id, messaggio) *// riceve un messaggio da qualsiasi processo.*

- Il parametro *id* della *receive()* è impostato sul nome del processo con il quale la comunicazione ha avuto luogo ed è recuperato da un campo dell’intestazione del messaggio ricevuto.

- La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione indiretta.

PRODUTTORE

```
pid cons = …
main(){
  Messaggio mes;
  do {
    produci (mes);
    send (cons, mes);
  while (!fine)
}
```

CONSUMATORE

```
main(){
  Messaggio mes;
  pid id;
  do {
    receive (id, mes);
    consuma (mes)
  while (!fine)
}
```

Schema di comunicazione indiretta

- La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione indiretta in cui il consumatore invia un messaggio di risposta al produttore.

```
PRODUTTORE

pid cons = …
main(){
  Messaggio mes, mes_ris;
  do {
    produci (mes);
    send (cons,mes);
    receive(cons, mes_ris);
  while (!fine)
}
```

```
CONSUMATORE

main(){
  Messaggio mes, mes_ris;
  pid id;
  do {
    receive (id,mes);
    consuma (mes);
    send(id, mes_ris);
  while (!fine)
}
```

Schema di comunicazione indiretta

- Con la comunicazione indiretta, i messaggi sono trasferiti mediante mailbox, o porte. Una mailbox è un’astrazione di un oggetto in cui i messaggi possono essere posti o prelevati dai processi.
- Ogni mailbox ha un identificativo univoco. Ad esempio, le code di messaggi POSIX utilizzano un valore di tipo string per identificare una mailbox.
- Le funzioni *send()* e *receive()* sono definite come segue:

send (mailbox, messaggio) // invia un messaggio alla mailbox

ricevere (mailbox, messaggio) // riceve un messaggio dalla mailbox

- Una mailbox può appartenere a un processo o al sistema operativo. Se la mailbox è parte dello spazio degli indirizzi del processo, allora si distingue tra il proprietario, che può ricevere solo messaggi, e l'utente che può solo inviare messaggi alla mailbox.
- Quando un processo proprietario di una mailbox termina, la mailbox è cancellata. Pertanto, qualsiasi processo che invia un messaggio a questa mailbox deve essere avvisato che la mailbox non esiste più.
- Al contrario, una mailbox che è di proprietà del sistema operativo non è collegata a un particolare processo.
- Il sistema operativo deve quindi fornire un meccanismo che consente a un processo di effettuare le seguenti operazioni:
  - **Creare una nuova mailbox.**
  - **Inviare e ricevere messaggi attraverso la mailbox.**
  - **Eliminare una mailbox.**
- Il processo che crea una nuova mailbox è proprietario di quella mailbox di default.

- Inizialmente, il proprietario è l’unico processo in grado di ricevere messaggi attraverso questa mailbox. Tuttavia, la proprietà e il privilegio di ricevere messaggi possono essere passati ad altri processi attraverso chiamate di sistema appropriate. Naturalmente, tale disposizione potrebbe prevedere più processi riceventi per ogni mailbox.

Sincronizzazione

- La comunicazione tra processi avviene attraverso chiamate di sistema *send()* e *riceive()*. Ci sono diverse opzioni per l'implementazione delle due funzioni.
- Lo scambio di messaggi può essere sia sincrono (bloccante) che asincrono (non bloccante).
- ***Send sincrona (bloccante)*.** Il processo mittente invia un messaggio e resta in attesa fino a quando il messaggio viene ricevuto dal processo destinatario o dalla mailbox.
- ***Send asincrona (non bloccante)***. Il processo mittente invia il messaggio e continua la sua esecuzione.
- ***Receive sincrona (bloccante)***. Il processo destinatario è bloccato fino a quando riceve un messaggio.
- ***Receive asincrona (non bloccante)****.* consente di continuare l'esecuzione del processo anche in assenza di messaggi.

- Sono possibili diverse combinazioni di send e receive.
- Quando sia la *send()* che *receive()* sono bloccanti, si ha uno schema detto ***rendez-vous*** *(appuntamento)*.
- La soluzione al problema ***produttore-consumatore*** è più semplice quando si usano *send()* e *receive()* bloccanti. In questa situazione, il mittente semplicemente chiama la *send()* e attende che il messaggio sia consegnato al destinatario o alla mailbox. Allo stesso modo, il ricevente quando chiama la *receive()*, si blocca fino a quando un messaggio è ricevuto.
- La *send()* non bloccante consente al processo che l'ha chiamata di continuare la sua esecuzione. Può risultare valida se il mittente non deve attendere una risposta dal processo ricevente.
- L'uso della *receive()* in modalità non bloccante può essere utile per analizzare, secondo un ordine stabilito, lo stato di più comunicazioni instaurate con vari client. In tal caso se su alcune connessioni non giungono messaggi si ha una poco efficiente forma di attesa attiva.

```
message_t msg_prodotto;
while (true) {
  < produce un messaggio >
  send(msg_prodotto);
}
```

Processo produttore con il message passing.

```
message_t msg_consumato;
while (true) {
  receive(msg_consumato);
  < consuma il messaggio >
}
```

Processo consumatore con il message passing.

Code di messaggi

- Se la comunicazione è diretta o indiretta, i messaggi scambiati dai processi sono posti in code temporanee. Principalmente, queste code possono essere implementate in due modi:
- ***capacità uno**.* La coda ha una lunghezza massima pari a uno. In questo caso, il mittente si deve bloccare finché il destinatario riceve il messaggio *(sistema senza buffering)*
- ***Capacità N***. La coda può contenere N messaggi. Se la coda non è piena quando viene inviato un nuovo messaggio, il messaggio viene inserito nella coda e il mittente può continuare l'esecuzione senza attendere. Se la coda è piena, il mittente si deve bloccare finché lo spazio è di nuovo disponibile nella coda. Il ricevente può leggere la coda fino a che contiene messaggi. Quando la coda è vuota il ricevente si blocca fino a quando nuovi messaggi sono disponibili *(sistema con il buffering automatico)*.

Code di messaggi POSIX

- Le code di messaggi POSIX sono identificate utilizzando nomi (string). Un processo deve conoscere il nome della coda e avere i permessi appropriati per inviare o ricevere messaggi dalla coda e anche fare altre operazioni su di essa.
- I programmi che utilizzano le code di messaggi POSIX su Linux devono essere compilati collegando la libreria in tempo reale **librt** utilizzando l'opzione del compilatore **-lrt** .
- I nomi delle funzioni iniziano con il prefisso, mq_ (abbreviazione di message queue)

.

mq_open , mq_close

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
mqd_t mq_open (const char *name, int oflag);
mqd_t mq_open (const char *name, int oflag,
    mode_t mode, struct mq_attr *attr);
```

- La funzione *mq_open()* è per l'apertura di una coda. Il primo parametro *name* specifica il nome della coda.
- Il secondo parametro è un flag che specifica la modalità di accesso alla coda. Può essere O_RDONLY per la ricezione di messaggi, O_WRONLY per l'invio di messaggi e O_RDWR per le operazioni sia di invio che di ricezione di messaggi sulla coda.
- Più valori possono essere assegnati a questo flag tramite l'operatore OR (|).

- È possibile specificare O_NONBLOCK per utilizzare la coda nella modalità non bloccante. Per impostazione predefinita, *mq_send()* è bloccante se la coda è piena e *mq_receive* è bloccante se non c'è alcun messaggio nella coda. Ma, se è specificato il valore O_NONBLOCK nel parametro *oflag*, la chiamata ritorna immediatamente con la variabile di sistema *errno* impostata al valore EAGAIN.
- Se si specifica O_CREAT come parte di *oflag*, è creata la coda, se già non esiste.
- Se è specificato O_CREAT in *oflag* allora deve essere utilizzata la seconda forma di *mq_open(),* con due parametri aggiuntivi. In tal caso, si possono specificare le autorizzazioni per la coda e il puntatore a una struttura di attributi *struct mq_attr* per la coda di messaggi. Se questo puntatore è NULL, viene creata una coda con attributi predefiniti.

```
struct mq_attr {
  long mq_flags; /* Flags: 0 or O_NONBLOCK */
  long mq_maxmsg; /* Max numero di messaggi */
  long mq_msgsize; /* Max dim. del messaggio (byte) */
  long mq_curmsgs; /* numero di messaggi correntemente
                      in coda */
};
```

- Se la chiamata *mq_open()* ha successo, viene restituito il descrittore della coda dei messaggi. Il descrittore di coda di messaggi è utilizzato per identificare la coda in chiamate successive.

- La chiamata ***mq_close*** è,

```
#include <mqueue.h>
int mq_close (mqd_t mqdes);
```

- La *mq_send()* è chiamata per inviare un messaggio alla coda specificata dal descrittore *mqdes*.

```
#include <mqueue.h>
int mq_send (mqd_t mqdes, const char *msg_ptr,
     size_t msg_len, unsigned int msg_prio);
```

- Il secondo parametro *msg_ptr* è un puntatore al messaggio da inviare.
- *Msg_len* è la dimensione del messaggio che deve essere minore o uguale alla dimensione specificata nel campo *mq_msgsize* nella struttura *mq_attr*.
- *Msg_prio* è un numero non negativo che specifica la priorità del messaggio. I messaggi sono inseriti nella coda in ordine decrescente di priorità. I messaggi con stessa priorità sono gestiti con politica FIFO.

- Se la coda è piena, *mq_send* blocca il processo fino a quando vi è spazio nella coda, a meno che il flag O_NONBLOCK sia abilitato per la coda di messaggi, nel qual caso mq_send ritorna immediatamente con *errno* impostato a EAGAIN .

mq_receive

```
#include <mqueue.h>
ssize_t mq_receive (mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned int *msg_prio);
```

- *mq_receive()* preleva un messaggio dalla coda specificata dal descrittore *mqdes*.
- Il messaggio più vecchio con la massima priorità viene prelevato dalla coda e passato al processo nel buffer puntato da *msg_ptr*.
- *msg_len* è la lunghezza del buffer in byte e deve essere maggiore o uguale della dimensione massima dei messaggi specificata nell'attributo *mq_msgsize* della coda.

- Se il puntatore *msg_prio* non è nullo, la priorità del messaggio ricevuto è memorizzato nell'intero puntato da esso.
- Il comportamento predefinito di *mq_receive()* è bloccante, se non vi è alcun messaggio nella coda. Tuttavia, se il flag O_NONBLOCK è settato per la coda, e la coda è vuota, *mq_receive()* ritorna subito con *errno* impostato a EAGAIN.
- In caso di successo, *mq_receive()* restituisce il numero di byte ricevuti nel buffer puntato da *msg_ptr*.

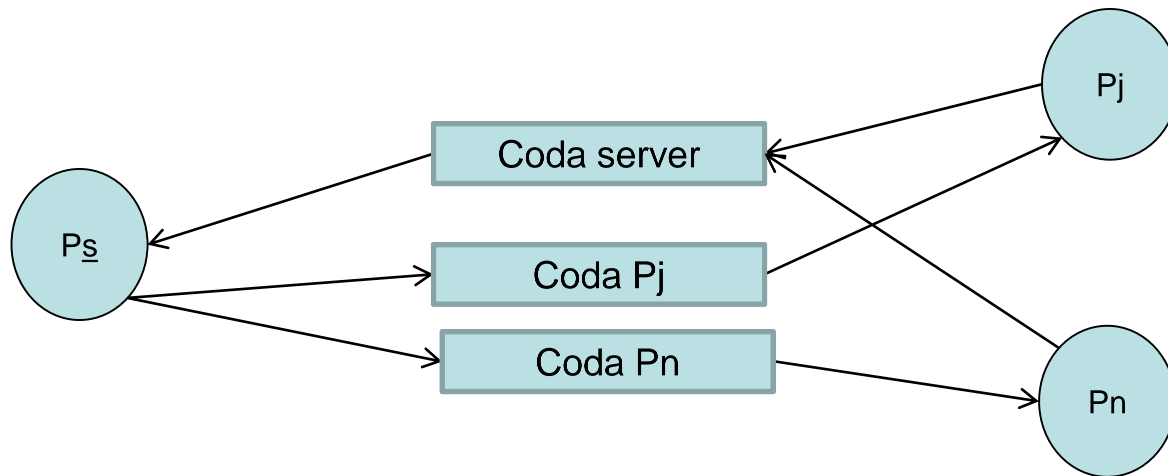
mq_unlink

```
#include <mqueue.h>
int mq_unlink (const char *nome_coda);
```

- *mq_unlink()* rimuove la coda con il nome *nome_coda*.

Esempio code di messaggi Posix

- Per mostrare un esempio d'uso delle code di messaggi Posix, realizziamo una semplice applicazione con architettura client/server con le seguenti specifiche:
  - un insieme di processi client richiede ad un processo server un numero seriale univoco di tipo intero
  - Il server risponde alla richiesta di un client inviando un numero diverso per ciascuna richiesta.
- Dalle specifiche richieste si deduce che si tratta di una **comunicazione indiretta**. Infatti, i client devono conoscere l'identità del server (in questo caso il nome della coda creata dal server) e il server non conosce le identità dei client. Questi, dovranno quindi inviare il loro identificativo al server, ponendolo in un campo del messaggio di richiesta. Inoltre, ogni client deve creare una propria coda dove potrà leggere la risposta del server.



- Per semplicità definiamo i messaggi di richiesta e risposta con lo stesso formato:

```
struct message {
  int pid;
  char text[64];
};
```

- Ciascun client inserirà nel campo *pid* il proprio identificativo (cioè il suo pid) e nel campo *text* un *nickname.* Il nickname sarà passato al lato client dell'applicazione come parametro d'avvio.
- Il server scriverà nel campo *text* il numero seriale da assegnare al client richiedente.
- Ciascun client creerà una propria coda assegnando ad essa un nome formato dal prefisso mq_ seguito dal suo pid (ad esempio mq_2018).
- Il codice dell'applicazione è mostrato di seguito. Per sperimentare l'applicazione client/server:
  - Aprire una finestra terminal (ad esempio una shell bash)
  - compilare i due programmi mq_server.c e mq_client.c utilizzando l'opzione del compilatore **-lrt** :
    ```
    gcc mq_server.c –o mq_server –lrt
    gcc mq_client.c –o mq_cient –lrt
    ```
  - Avviare il server in background inserendo nel comando il carattere &:
    ```
    ./mq_server &
    ```
  - Avviare il client (fornendo un nickname come parametro) in foreground:
    ```
    ./mq_client pedro
    ```

```
// Server  (file mq_server.c)
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <mqueue.h>
#define SERVER_QUEUE_NAME "/mq_server"
#define QUEUE_PERMISSIONS 0660
#define MAX_MESSAGES 10

struct message {
       int pid;
       char text[64];
} msg_send, msg_rcv;
```

```
int main (int argc, char *argv[]) {
  mqd_t qd_server, qd_client; // descrittori code
  long serial_number = 1; /* numero seriale da inviare al
                             client */
  char client_queue_name[16]; /* nome coda del client */
  struct mq_attr attr; // attributi della coda
  attr.mq_flags = 0; // bloccante
  attr.mq_maxmsg = MAX_MESSAGES; /* numero massimo di
                                    messaggi della coda */
  attr.mq_msgsize = sizeof(msg_rcv);
  attr.mq_curmsgs = 0;

  printf ("Server: Benvenuto!\n");
  qd_server = mq_open (SERVER_QUEUE_NAME, O_RDONLY |
                  O_CREAT, QUEUE_PERMISSIONS,&attr);
```

```
  while (1) {
    /* preleva dalla coda il messaggio più vecchio con
       piu' alta  priorita‘ */
    mq_receive (qd_server, (char*)&msg_rcv,
       sizeof(msg_rcv), NULL);
    printf ("Server: messaggio ricevuto dal client %d,
       %s.\n",msg_rcv.pid,msg_rcv.text);
    sprintf (client_queue_name, "mq_%d", msg_rcv.pid);
    /* invia il messaggio di risposta al client */
    qd_client = mq_open (client_queue_name, O_WRONLY);
    sprintf(msg_send.text,"Benvenuto client %d, il tuo
       numero e' %ld", msg_rcv.pid, serial_number);
    mq_send (qd_client, (const char *)&msg_send,
       sizeof(msg_send), 0);
    printf ("Server: risposta inviata al client.\n");
    serial_number++; // incrementa il serial_number
  }
}
```

```c
// Client  (file mq_ client.c)
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <mqueue.h>

#define SERVER_QUEUE_NAME "/mq_server"
#define QUEUE_PERMISSIONS 0660
#define MAX_MESSAGES 10

struct message {
       int pid;
       char text[64];
} msg_send, msg_rcv;
```

```
int main (int argc, char *argv[]) {
  char client_queue_name[16];
  mqd_t qd_server, qd_client; /* descrittori code
   crea coda client per ricevere i messaggi dal server */
  sprintf (client_queue_name, "mq_%d", getpid ());
  struct mq_attr attr;
  attr.mq_flags = 0;
  attr.mq_maxmsg = MAX_MESSAGES;
  attr.mq_msgsize = sizeof(msg_send);
  attr.mq_curmsgs = 0;
  qd_client = mq_open (client_queue_name,
       O_CREAT|O_RDONLY,QUEUE_PERMISSIONS,&attr);
  /* apre la coda del server per inviare il proprio
     identificatore (pid) */
  qd_server = mq_open (SERVER_QUEUE_NAME,O_WRONLY);
  printf ("Richiedi un numero (Premi <INVIO>):");
  char in_buf [10];
```

```
msg_send.pid=getpid();
if (argc > 1)
  strcpy(msg_send.text,argv[1]); // nickname
else {
  strcpy(msg_send.text,"Pietro"); // default nickname
}
while (fgets (in_buf, 2, stdin)) {
  // invia messaggio al server
  mq_send (qd_server, (const char *)&msg_send, sizeof
     (msg_send), 0);
  // riceve risposta dal server
  mq_receive (qd_client, (char *)&msg_rcv, sizeof
     (msg_rcv), NULL);
  // visualizza messaggio ricevuto dal server
  printf ("Client: messaggio dal server: %s\n\n",
     msg_rcv.text);
  printf ("Richiedi un numero  (Premi <INVIO>): ");
}
```

```
  mq_close (qd_client);
  mq_unlink (client_queue_name); // elimina coda
  printf ("Client: ciao\n");
}
```