PVM – Parallel Virtual Machine

Charakteristika

- Původně pro C a Fortran, dnes i třeba JPVM
- Univerzální výpočetní model pro heterogenní distribuované výpočetní prostředí
- Abstrakce několika různých prostředí
    - Různé OS
    - Různý HW
    - PVM poskytuje rozhraní, které je umožní využívat jako jeden paralelní počítač
        - Lze vytvořit i virtuální síť na jednom počítači
- Místo vláken procesy, které tvoří distribuovanou aplikaci

```
hello_other.c
include "pvm3.h"
main()
{
    int ptid;
    char buf[100];
    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);

    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, 1);
    pvm_exit();
    exit(0);
}
```

hello.c

```
#include <stdio.h>
#include "pvm3.h"

main()
{
    int cc, tid;
    char buf[100];

    printf("I'm t%x\n", pvm_mytid());

    cc = pvm_spawn("hello_other",
                (char**)0, 0, "", 1, &tid);

    if (cc == 1) {
        cc = pvm_recv(-1, -1);
        pvm_bufinfo(cc, (int*)0,
                        (int*)0, &tid);
        pvm_upkstr(buf);
        printf("from t%x: %s\n", tid, buf);

    } else
        printf("can't start hello_other\n");

    pvm_exit();
    exit(0);
}
```

http://www.cs.hmc.edu/qref/pvm/pvm.html

Konzole PVM

- spustí se příkazem *pvm*
  - pvm>
  - opustí se příkazem *quit*
    - PVM ale stále poběží
  - *halt* – zastaví pvm na všech počítačích kde běží a ukončí všechny běžící procesy

- přidání počítačů dostupných pro výpočet
  - *add* <hostname>
  - odstranění se provede *delete* <hostname>

- *conf* – konfigurace pvm
- *spawn* – spustí proces, viz dále
  - spawn –count executable
  - počet úloh, když není, předpokládá se jedna

- *echo, help, id, jobs, mstat <host tid>, reset, setenv, trace* a také funguje přesměrování vstupu a výstupu ->, ->>

- v PVM API existují funkce s podobným, či téměř stejným názvem, ale mají prefix *pvm_*
- Konfigurace (pod Linuxem)
  - $HOME/rhosts – seznam počítačů a uživatelských jmen
  - proměnné – PVM_ROOT, PVM_ARCH, XPVM_ROOT (graf. konzole a monitor PVM)
  - $HOME/pvm3/bin/PVM_ARCH
    - $HOME/pvm3/bin/RS6K – IBM/RS6000
    - $HOME/pvm3/bin/LINUX – i486 & Linux
    - pvmgetarch

Spawn

- jako příkaz v konzoli spustí primární proces
- pvm_spawn spustí několik procesů podle zadaného programu
    - vrací identifikátory procesů
    - lze buď určit, kde se mají procesy spustit,
    - nebo to lze nechat na rozhodnutí PVM
    - má-li proces PVM běžet na různých platformách, je třeba, aby programátor dal k dispozici verze pro všechny tyto platformy
- procesy se po vytvoření neznají
    - proces, který je vytvořil, je musí seznámit

- int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)
    - numt – počet spuštěných procesů
    - task – spustitelný soubor
    - argv – parametry
    - flag – možnosti spuštění
        - PvmTaskDefault – PVM si rozhodne, kde spustit
        - PvmTaskHost – where bude obsahovat adresu, kde spustit
        - PvmTaskArch – where bude obsahovat typ architektury/platformy
        - Existují i další jako PvmTaskDebug, PvmTaskTrace, PvmMppFront, PvmHostCompl
    - where – kde spustit proces, ignoruje se, pokud nejsou příslušně nastaveny možnosti spuštění
    - ntask – počet procesů ke spuštění
    - tids – identifikátory spuštěných procesů

Komunikace

- probíhá pomocí zasílání zpráv
- adresu procesu, v terminologii PVM tasku, tvoří TID – Task ID

- zaslání zprávy od procesu A procesu B
  - A zavolá pvm_initsend, aby vyprázdnil buffer k odeslání a nastavil kódování zprávy
  - A zavolá některou z pvm_pack funkcí, která vloží data k odeslání do bufferu
    - int pvm_pkint(int *ip, int nitem, int stride)
      - result – 0 OK, <0 chyba
      - ip – pointer na pole hodnot typu int
      - nitem – počet hodnot v poli
      - stride – krok, např. stride = 2 znamená, že se do bufferu uloží každé druhé číslo
  - jakmile A umístil do bufferu všechny data, zavolá pvm_send
    - int pvm_send(int tid, int msgtag)
      - result – chybový kód
      - tid – task id příjemce
      - msgtag – dodatečná informace, kterou může příjemce využít – např. id zprávy, ze které odvodí strukturu dat ve zprávě
        - mělo by být >=0

  - existuje i pvm_psend, která zabalí a odešla data najednou
  - existují i funkce pro přímou manipulaci s bufferem
    - pvm_mkbuff, pvm_freebuf

- chce-li proces B přijmout data od procesu A
    - B zavolá pvm_recv
        - int pvm_recv(int tid, int msgtag)
            - result – chybový kód
            - tid – task id příjemce
                - -1 znamená přijmi zprávu od kohokoliv
            - msgtag – popis zprávy, která se má přijmout
                - může být -1, má-li se ignorovat

        - pvm_recv/pvm_precv je blokující operace
            - pvm_trecv – čeká na zprávu pouze do vypršení timeoutu
            - pvm_nrecv – neblokující operace
            - pvm_probe – testuje přítomnost nové zprávy

    - po přijetí zprávy B použije funkce pvm_unpack analogicky k A, který použil pvm_pack

Kódování zprávy

```
aligned = record
  oneByte: byte;
  oneInt:  integer;
end;

nonaligned = packed record
  oneByte: byte;
  oneInt:  integer;
end;
```

při optimalizaci kódu, u typu aligned začíná oneInt na offsetu, který se rovná sizeof(integer)

- o non-aligned znamená offset vždy jeden byte
- o => nelze jen tak použít přímou reprezentaci dat v paměti, když procesy nejsou identické a platformy se liší

- big-endian vs. little-endian
- int je 16, 32, nebo 64-bitový?
  - závisí na překladači

- řešením je jednotné kódování zprávy pro všechny platformy, jejichž reprezentace dat v paměti by se jinak mohla lišit

- kódování se nastavuje pomocí pvm_initsend
  - int bufid = pvm_initsend(int encoding)
    - bufid – id bufferu, nebo chyba pokud <0
    - encoding – kódování zprávy

      - PvmDataDefault – XDR
        - eXternal Data Representation

      - PvmDataRaw
        - programátor si sám musí pohlídat správné kódování a dekódování zprávy v heterogenním prostředí
        - lze ušetřit strojový čas, který by jinak padl na kódování

- PvmDataInPlace
    - do (message) bufferu se vkládají pouze pointry, ne data
    - data se zkopírují z paměti teprve až při odeslání
    - data se normálně kopírují alespoň nadvakrát
        1. do message bufferu
        2. do zprávy
        3. – n. zajištění izolace procesů na úrovni OS
        - Způsob jak ušetřit režii spojenou s kopírováním
    - stride musí být vždy 1

XDR

- IETF standard (Internet Engineering Task Force)
- je předepsáno, jak se budou ukládat standardně definované datové typy
    - boolean
    - int (32 bit integer)
    - hyper (64 bit integer)
    - float
    - double
    - enumeration
    - structure
    - string
    - fixed length array
    - variable length array
    - union
    - opaque data

- všechny ostatní datové typy (opaque data) se ukládají jako sekvence bytů
- mimo PVM se používá např. u RPC (Remote Procedure Call)
- standardní operace jsou encoding (do XDR) a decoding (z XDR)
    - PVM používá termíny packing a unpacking
        - XDR totiž není jediná volba – viz pvm_initsend

    - DCOM používá marshalling
        - „objektová verze“ RPC
            - Bylo nutné rozšířit i možnosti standardizovaného XDR
            - Umožňuje Handler Marshalling – možnost vstoupit do procesu marshallingu

    - Další příbuzný termín je serializace

- Existují i další standardy
    - SDXF (Structured Data eXchange Format)
    - Ale třeba i XML je nezávislé na platformě

Dynamické skupiny procesů

- pokud jsou procesy spřízněny činností, kterou vykonávají, lze je sloučit do skupiny
    - tím ovšem nepřijdou o své tid
- provádí se voláním funkcí pvm_joingroup
    - int inum = pvm_joingroup(char *group)
    - skupinu lze pojmenovat
    - pokud skupina neexistuje, je vytvořena

  - proces může být členem několika skupin
  - vrací pořadové číslo procesu ve skupině
    - přiděluje se nejnižší možné číslo

- proces opustí skupinu funkcí pvm_lvgroup
  - vznikne mezera v pořadových číslech ve skupině, dokud se ke skupině nepřidá další proces

- pvm_getinst – vrací číslo instance procesu ve skupině
- pvm_gettid – vrací tid procesu podle jeho skupiny a čísla jeho instance ve skupině
- pvm_gsize – vrací velikost skupiny

Synchronizace

- pvm_bcast – broadcast zprávy všem procesům ve skupině
- pvm_mcast – multicast zprávy procesům identifikovaným polem tid
- pvm_barrier – proces se zastaví, dokud všechny procesy ve skupině nezavolají tuto funkci
- pvm_sendsig – zašle specifikovaný signál danému procesu – jak to udělají ve Windows?

- pvm_gather
  - daný člen skupiny přijme zprávy od všech ostatních členů skupiny
  - zprávy jsou sloučeny do jednoho pole
  - všichni musí zavolat pvm_gather
  - daný člen skupiny je root skupiny, identifikován svým číslem instance ve skupině (parameter funkce)

    - funkce není blokující a pokud některý člen opustí skupinu dříve než root zavolá pvm_gather, může dojít k chybě
- pvm_scatter – umožňuje odeslání pole členům skupiny, jako kdyby se pole postupně rozdělilo a každému z nich extra poslala jenom jedna část

- pvm_reduce
    - umožňuje provést globální operaci, jako je třeba nalezení minima nebo spočítání průměru, nad daty, která jsou distribuována přes procesy ve skupině
    - každý proces zavolá pvm_reduce nad svou částí dat
    - výsledek se odešle rootu skupiny

Kritická sekce & spol.

- Pokud chce proces zaručit exkluzivní přístup k prostředkům, které spravuje, pak je to pouze otázka komunikačního protokolu procesů
    - Vždy povolí manipulaci s daty pouze jednomu z procesů

- Pokud je ale z nějakého důvodu nutné vybrat pouze jeden proces, existují na to speciální distribuované algoritmy
    - Leader election
    - Token
    - Distribuované vyloučení
    - Hlasování

Ukončení výpočtu

- pvm_kill – násilné ukončení procesu s daným id
- pvm_exit – dobrovolné ukončení výpočtu procesem
- proces se ukončí pouze v rámci PVM, pro OS stále běží

Výkonnostní hlediska

- granularita procesů
  - poměr mezi objemem komunikace, tj. velikostí dat, a operacemi nad daty prováděnými
  - je třeba vyvážit využití procesoru a komunikačních linek – efektivita paralelizace

- granularita zpráv
  - velké zprávy ušetří počet volání kom. funkcí
  - malé zprávy se doručují rychleji
    - ale až příliš malé zprávy zase nejsou odesílány ihned – Nagleův algoritmus
  - zasílání zpráv se může překrývat v čase
  - specifické pro daný program

- funkční vs. datový paralelismus
  - aplikace PVM může zároveň běžet na vektorovém počítači, který je pro určitý druh operaci efektivnější
  - další část PVM aplikace zase může běžet na grafické stanici, které dokáže data zobrazovat dostatečně rychle

- když máme omezený počet procesů na různě rychlých strojích
    - práci lze přidělovat v malých objemech, tak, aby rychlejší procesy nečekaly na pomalejší

- přenositelnost – použít specifické funkce, které jsou sice rychlé/pohodlné/jinak výhodné, ale platformě závislé?

- Load-Sharing, Load-Balancing a Load-Redistribution
    - 11. přednáška

Řízení procesů

- každý proces má svého démona
    - zprostředkovává doručování zpráv
    - je zodpovědný za spolehlivý přenos a doručení zprávy
    - zprávu k odeslání vždy přebere démon příslušný k procesu a z tid zjistí, kam ji poslat
    - provádí un/packing

- existuje jeden master daemon
    - centralizace
    - možnost přetížení
    - pokud selže, selže celá aplikace

- součástí konfigurace PVM je hostfile, který určuje síťové uzly, na kterých běží démoni
    - lze ho měnit buď z konzole PVM
    - nebo programově
        - pvm_addhosts
        - pvm_delhosts

Informační funkce

- pvm_mytid
- pvm_parent
- pvm_tidtohost
- pvm_config
- pvm_tasks
- pvm_getopt – lze je měnit pomocí pvm_setopt

Pokročilé funkce

- pvm_reg_hoster
    - nahradí funkci, která se stará o spouštění démonů
    - když master daemon obdrží zprávu DM_ADD, nespustí nového démona, ale předá řízení definované funkci
    - addhost

- pvm_reg_tasker
    - zaregistruje volajícího jako spouštěč PVM procesů
    - když master daemon obdrží zprávu DM_EXEC, nespustí nový proces (i.e. fork, exec na UNIXu, Linuxu), ale předá slovo zaregistrovanému, který se o to postará a master daemonu už dá jenom vědět výsledek operace
    - spawn
- pvm_reg_rm
    - zaregistruje volajícího jako správce zdrojů
    - zaregistrovaný např. obsluhuje požadavky na služby od ostatních démonů, dostává hlášení o selhání systémů…
    - umožňuje definovat nový plánovač

The Linda System

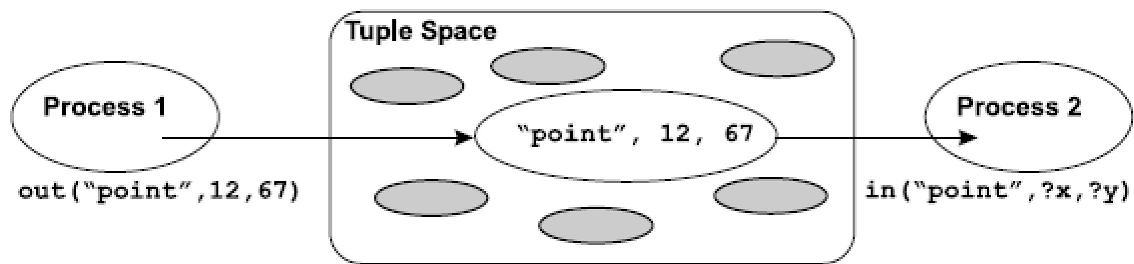
- z hlediska programu jenom další rozšíření knihovny
- alternativa ke sdílené paměti a zasílání zpráv
- funguje na konceptu n-tic (tuples)
  - záznamy s typovanými složkami
- poskytuje sdílenou paměť, kde jsou n-tice nesoucí informace (tuple-space)

- asociativní paměť
  - asociativní adresování s pomocí tzv. antituples/templates
    - část n-tice má definovaný obsahpoužívají se k nalezení n-tic, které mají stejný obsah na té samé části
    - zbytek n-tice se nazývá wildcard
    - jako filemask, kdy jedné masce odpovídá n souborů

- minimální počet základních operací
  - in – atomické a destruktivní čtení n-tice
  - rd – nedestruktivní čtení
  - out – zápis n-tice
  - eval – vytvoří nový proces k testu n-tic a zápisu výsledku

- zbavuje programátora potřeby adresovat procesy
- koordinační jazyk, který je zodpovědný výhradně za koordinaci a realizaci komunikace ponechává na hostitelském prostředí
- http://wcat05.unex.es/Documents/Wells.pdf

- Proč jim nestačí network-wide named blackboards?




```
// An Entry class
public class SpaceEntry implements Entry
{
	public final String message = "Hello World!";
	public Integer count = 0;

	public String service() {
		count = new Integer(count.intValue()+ 1);
		return message;
	}

	public String toString() {
		return "Count: " + count.toString();
	}
}
```

```
// Hello World! server
public class Server
{
      public static void main(String[] args)
      {
           try {
               SpaceEntry entry = new SpaceEntry();
// Create the Entry object
               JavaSpace space = (JavaSpace)space();
// Create an Object Space
// Register and write the Entry into the Space
               space.write(entry, null,
                           Lease.FOREVER);
// Pause for 10 minutes and then retrieve the
// Entry and check its state.
               Thread.sleep(10*60*1000);
               SpaceEntry e = space.read(new
                                 SpaceEntry(), null,
                                 Long.MAX_VALUE);
               System.out.println(e);
           } catch (Exception e) {}
      }
}

// Client
public class Client
{
      public static void main(String[] args)
      {
           try {
               JavaSpace space =
                             (JavaSpace) space();
               SpaceEntry e = space.take(new
                               SpaceEntry(), null,
                               Long.MAX_VALUE);
               system.out.println(e.service());
               space.write(e, null, Lease.FOREVER);
           } catch (Exception e) {}
      }
}
```

```
//Farmer
#include <stdio.h>
#include <pvm3.h>

main() {
  int mytid;       /*  identifikator
                       procesu sef  */
  int tids[32];    /*  identifikatory
                       procesu delnici */
  int nproc;       /*  pocet delniku */
  int m;           /*  kolik prvocisel se
                       ma zjistit */
  int* rslt;       /*  ukazatel na pole
                       vysledku */
  int last_rslt;
  int worker_cnt = -1894;   /* nastaveni
           pro kontrolu smysluplnosti */
  int rslt_num;    /*  kolik prvocisel uz
                       se naslo */
  int cnt_sent = 0;/* kolik prikazu
                       odeslano */
  int cnt_received = 0;/*  kolik vysledku
                            prijato */
  int i, k, tid;  /*  pomocne promenne */
  int end_param = 0;
  int vytvoreno = 0;  /*  pocet uspesne
                  vytvorenych delniku */
  int* tidc;       /* pomocny ukazatel pro
                      vytvareni delniku */
```

```
while (! end_param) {
  printf("Zadej mezni hodnotu pro
          zjistovani prvocisel:");
  scanf("%d", &m);
  printf("Zadej pocet delniku
          (1 az 32):");
  scanf("%d", &nproc);
  if (nproc > m)
    printf("Prilis mnoho delniku -
            budou se flinkat!");
  else end_param = 1;
}

/* inicializace pole pro vysledky
   vypoctu */
rslt = (int *)malloc (m*sizeof(int));
*rslt = 1;    /* prvni prvocislo vime
                 rovnou */
rslt++;
*rslt = 2;    /* druhe prvocislo take
                 vime rovnou */
rslt++;
rslt_num = 2;

mytid = pvm_mytid();
```

```
/* vytvoreni delniku */
tidc = tids;
while (nproc>vytvoreno) {
 if (pvm_spawn("worker", (char**)0,
                0, "", 1, tidc)) {
    printf("Vytvoren delnik %d.\n",
                                *tidc);
    vytvoreno++;
    tidc++;
 } else {
    printf("Chyba pri vytvářeni delnika
            Cislo chyby %d\n", *tidc);
 }
}

/* uvodni komunikace sefa s delniky
(registrace prac. sil) */
pvm_initsend(PvmDataDefault);
            /* inicializace bufferu */
pvm_pkint(&mytid, 1, 1);
            /* ulozeni identifikace */
pvm_mcast(tids, nproc, 4);
       /* broadcast zpravy s kodem 4 */
       /*              vsem delnikum */
       /* cekani na odezvu od delniku*/
for(i=0; i<nproc; i++) {
  pvm_recv(-1, 5);
              /* 5 je kod zpravy */
  pvm_upkint(&tid, 1, 1);
              /* cislo delnika */
  pvm_upkint(&worker_cnt, 1, 1);
```

```
                  /* pocet zpracovanych
                     kousku prace delnika */

    printf("Delnik cislo %d k praci
            pripraven!\n ", tid);
  }

  /* pocatecni ukolovani delniku */
  k = 3;
  for(i=0; i<nproc; i++) {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&k, 1, 1);
    pvm_send(tids[i], 6);
    cnt_sent++;
    k += 2;
  }

  /* kdo dokoncil, dostane dalsi ukol */
  while (k <= m) {
    pvm_recv (-1, 7);
                 /* cekani na vysledek */
    cnt_received++;
    pvm_upkint (&tid, 1, 1);
    pvm_upkint (rslt, 1, 1);
        /* kopirovani do pole vysledku */
    pvm_upkint(&worker_cnt, 1, 1);
        /* pocet zpracovanych
           kousku prace delnika */

    if (*rslt>0) {
      rslt++; rslt_num++;
                    /* kladna hodnota
                       je to prvocislo */
```

```
    }



    /* zadani dalsiho ukolu */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&k, 1, 1);
    cnt_sent++;
    pvm_send(tid, 6);
    k += 2;
  }

  /* dobeh rozdelanych ukolu */
  while (cnt_sent > cnt_received) {
    pvm_recv (-1, 7);
                /* cekani na vysledek */
    cnt_received++;
    pvm_upkint (&tid, 1, 1);
    pvm_upkint (rslt, 1, 1);
    pvm_upkint (&worker_cnt, 1, 1);
    printf("%d  %d  %d  %d \n", cnt_sent,
           cnt_received, worker_cnt, tid);

    if (*rslt>0) {
      rslt++; rslt_num++;
                  /* kladna hodnota
                     je to prvocislo */
    }
  }
```

```
  /* uvolneni delniku */
  for(i=0; i<nproc; i++) {
    k = -m;//záporná hodnota => konči
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&k, 1, 1);
    pvm_send (tids[i], 6);

    pvm_recv (tids[i], 7);
              /* cekani na ukonceni */
    pvm_upkint (&tid, 1, 1);
    pvm_upkint (&last_rslt, 1, 1);
    pvm_upkint (&worker_cnt, 1, 1);
    printf("Delnik %d konci s hodnotou %d
            Vykonal prace: %d \n ",
      tids[i], last_rslt, worker_cnt);
  }

  /* tisk vysledku */
  rslt -= rslt_num;
  printf("Seznam nalezenych prvocisel:");
  for (i = 0; i < rslt_num; i++)
    printf("%d \n ", rslt[i]);

  /* ukonceni cinnosti v PVM */
  pvm_exit();

}
```

```
//Worker
#include "pvm3.h"

main() {
  int mytid;       /*  identifikator
                       procesu delnik  */
  int chief_tid;   /*  identifikator
                       procesu sef  */
  int x_num;       /*  zkoumane cislo */
  int cnt_work = 0;/*  kolik prvocisel
                       hledal */
  int k;           /*  pomocna promenna */
  mytid = pvm_mytid();

  /* cekani na zpravu od sefa */
  pvm_recv(-1, 4);
  pvm_upkint(&chief_tid, 1, 1);

  /* kontrola zda je sef ten pravy –
     totozny s otcem */
  if (chief_tid == pvm_parent()) {
    /* odpoved sefovi */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&mytid, 1, 1);
    pvm_pkint(&cnt_work, 1, 1);
    pvm_send(chief_tid, 5);
  } //else měl by čekat dál na zprávu 4
  /* delnik pracuje */
  while (1) {  /* !! zadne odbory */
    pvm_recv (chief_tid, 6);
    pvm_upkint (&x_num, 1, 1); /
```

```
    if (x_num < 0) {
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&mytid, 1, 1);
      pvm_pkint(&x_num, 1, 1);
      pvm_pkint(&cnt_work, 1, 1);
      pvm_send(chief_tid, 7);
      break;
               /* zaporna hodnota
                  znamena konec prace */
    }
       /* vlastni vypocet - pokud x_num
       je prvocislo, vrati se nezmenene,
       v opac. pripade se vrati zaporna
       hodnota */
    for (k=3; k < x_num; k+=2) {
      cnt_work++;
      if (!(x_num % k)) {
             /* test zbytku po deleni */
        x_num = -x_num;
             /* neni to prvocislo */
        break;
      }
    }
    /* odeslani vysledku zkoumani */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&mytid, 1, 1);
    pvm_pkint(&x_num, 1, 1);
    pvm_pkint(&cnt_work, 1, 1);
    pvm_send(chief_tid, 7);
  }
  /* ukonceni cinnosti */
  pvm_exit();
}
```