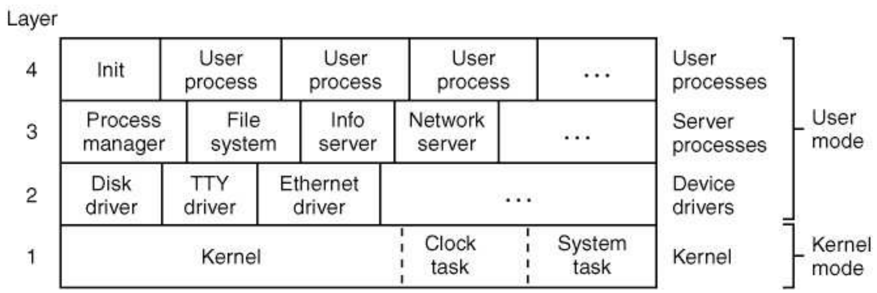
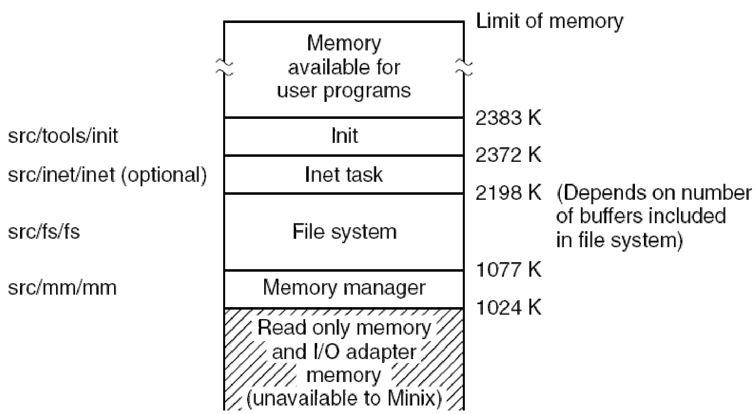
Architettura di Minix



Separazione di responsabilità

- *policy* (quale processo mettere in memoria) PM
- *mechanism* (manipolazione registri CPU) system task (kernel)

Memory layout



Gestione della memoria

Minix non usa la paginazione

Allocazione contigua

List of holes ordered by size; best fit

Un programma deve dichiarare quanta memoria desidera

chmem – change memory allocation

EXAMPLES

chmem =50000 a.out # Give a.out 50K of stack space
chmem -4000 a.out # Reduce the stack space by 4000 bytes
chmem +1000 file1 # Increase each stack by 1000 bytes

DESCRIPTION

... If the combined stack and data seg- ment growth exceeds the stack space allocated, the program will be terminated.

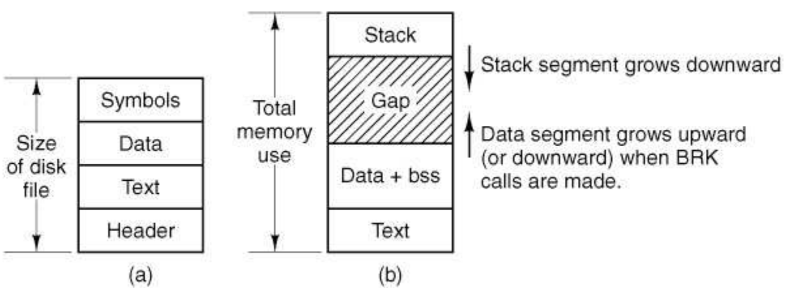
It is therefore important to set the amount of stack space carefully. If too little is provided, the program may crash. If too much is provided, memory will be wasted,

Shared text

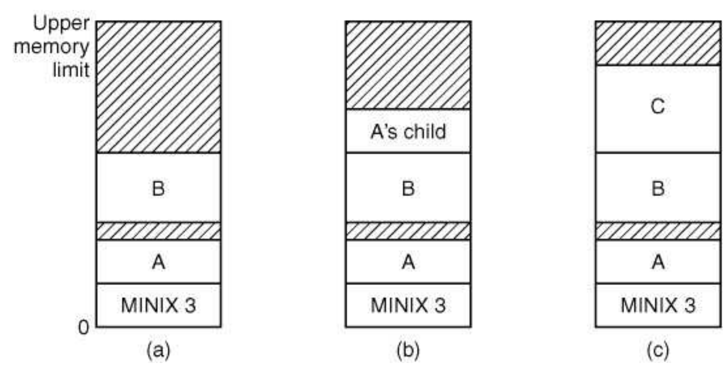
Due maniere di linkare:

- combined I and D space
- separate I and D space

Relazione fra file binario e layout di memoria del processo



Allocazione di memoria



(a) Prima. (b) Dopo una *fork*. (c) Dopo che il figlio fa una *exec*.

Message type	Input parameters	Reply value
fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
waitpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time
pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
sigreturn	Context	Status
getuid	(none)	Uid, effective uid
getgid	(none)	Gid, effective gid
getpid	(none)	PID, parent PID

setuid	New uid	Status
setgid	New gid	Status
setsid	New sid	Process group
getpgrp	New gid	Process group
time	Pointer to place where current time goes	Status
stime	Pointer to current time	Status
times	Pointer to buffer for process and child times	Uptime since boot
ptrace	Request, PID, address, data	Status
reboot	How (halt, reboot, or panic)	(No reply if successful)
svrctl	Request, data (depends upon function)	Status
getsysinfo	Request, data (depends upon function)	Status
getprocnr	(none)	Proc number
memalloc	Size, pointer to address	Status
memfree	Size, address	Status
getpriority	Pid, type, value	Priority (nice value)
setpriority	Pid, type, value	Priority (nice value)
gettimeofday	(none)	Time, uptime

Strutture dati del PM

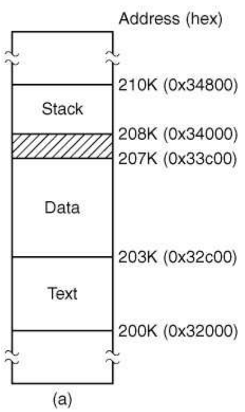
- Process table
- Hole table

La process table tripartita

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Time when process started CPU time used Children's CPU time Time of next alarm Message queue pointers Pending signal bits Process id Various flag bits	Pointer to text segment Pointer to data segment Pointer to bss segment Exit status Signal status Process id Parent process Process group Real uid Effective uid Real gid Effective gid Bit maps for signals Various flag bits	UMASK mask Root directory Working directory File descriptors Effective uid Effective gid System call parameters Various flag bits

kernel, PM, FS

Un processo in memoria



	Virtual	Physical	Length
Stack	0x8	0xd0	0x2
Data	0	0xc8	0x7
Text	0	0xc8	0

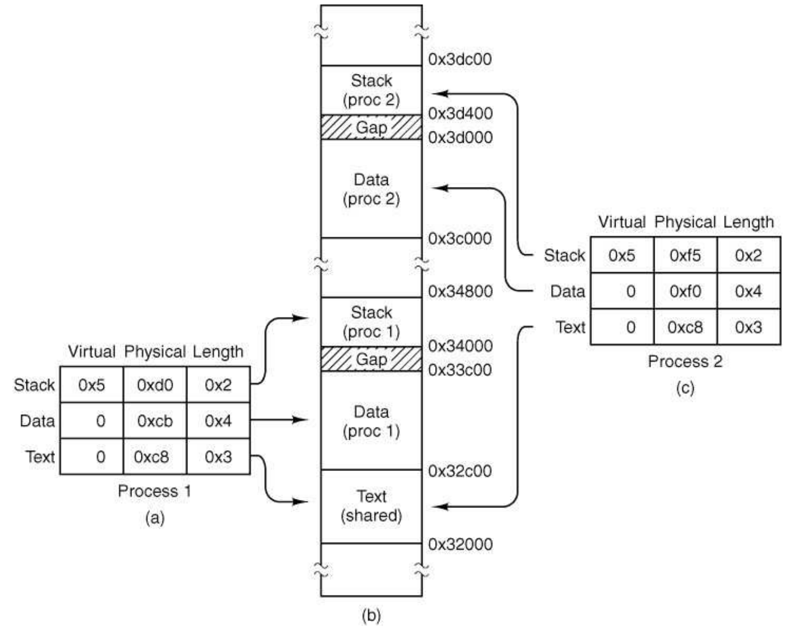
(b)

	Virtual	Physical	Length
Stack	0x5	0xd0	0x2
Data	0	0xcb	0x4
Text	0	0xc8	0x3

(c)

(b) I e D combinati. (c) I e D separati.

I e D separati



La chiamata di sistema fork(2)

1. Check to see if process table is full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a PID for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

La chiamata di sistema exec(2)

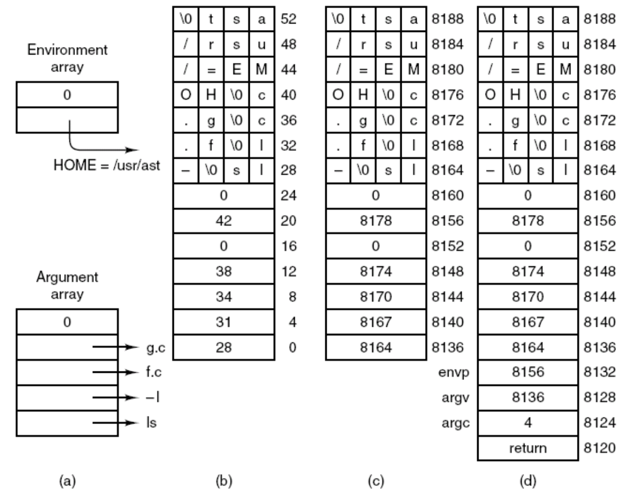
1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Esempio di exec

`ls -l f.c g.c`

`execve("/bin/ls", argv, envp);`

Esempio: esecuzione di `ls -l f.c g.c`



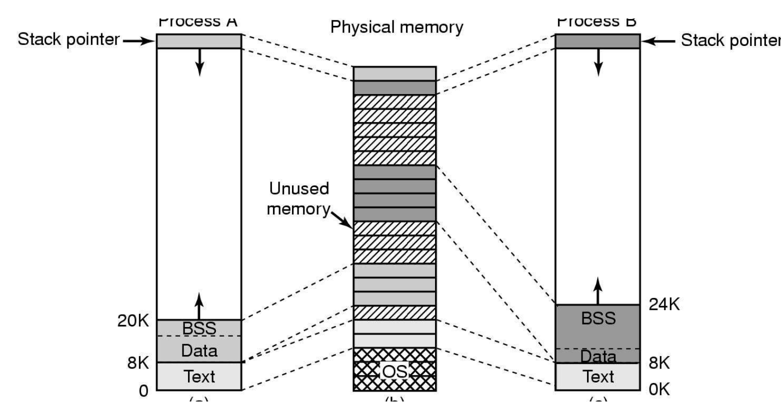
(a) array passati a *execve*. (b) Stack costruito da *execve*. (c) Stack dopo la rilocazione. (d) Stack come appare al main all'inizio

`crtso` C runtime start-off routine

```
push  ecx          ! push environ
push  edx          ! push argv
push  eax          ! push argc
call  _main        ! main(argc, argv, envp)
push  eax          ! push exit status
call  _exit
hlt                ! force a trap if exit fails
```

`main(int argc, char** argv, char** envp)`

Lo spazio di indirizzamento di un processo in UNIX



Come funziona malloc(3)?

`void *sbrk(int increment)`
sbrk increments the program's data space by increment bytes. Calling sbrk with an increment of 0 can be used to find the current location of the program break.

On success, sbrk returns a pointer to the start of the new area. On error, -1 is returned, and errno is set to ENOMEM.

— manuale di sbrk(2)

Una semplice implementazione di Malloc(3)

(adattata da *Inside memory management* di Jonathan Bartlett)

Var globali:

```
int has_initialized = 0;

void *managed_memory_start;

void *last_valid_address;
```

Inizializzazione

```
#include <unistd.h>   /* Include the sbrk function */

void malloc_init() {
  /* grab the last valid address from the OS */
  last_valid_address = sbrk(0);

  /* we don't have any memory to manage yet */
  managed_memory_start = last_valid_address;

  /* Okay, we're initialized and ready to go */
  has_initialized = 1;
}
```

Memory control block

```
struct mem_control_block {
  int is_available;
  int size;
};
```

La funzione free(3)

```
void free(void *firstbyte) {
  struct mem_control_block *mcb;

  /* Backup from the given pointer to find the
   * mem_control_block
   */
  mcb = firstbyte - sizeof(struct mem_control_block);

  /* Mark the block as being available */
  mcb->is_available = 1;

  /* That's It!  We're done. */
  return;
}
```

Outline di malloc(3) I

```
void *malloc(long numbytes) {

  /* Holds where we are looking in memory */
  void *current_location;

  /* This is the same as current_location, but cast to a
   * memory_control_block
   */
  struct mem_control_block *current_location_mcb;

  /* This is the memory location we will return.  It will
   * be set to 0 until we find something suitable
   */
  void *result;

  /* Initialize if we haven't already done so */
  if (!has_initialized) {
    malloc_init();
  }

  /* The memory we search for has to include the memory
   * control block, but the users of malloc don't need
   * to know this, so we'll just add it in for them.
```

Outline di malloc(3) II

```
   */
  numbytes = numbytes + sizeof(struct mem_control_block);

  /* Set result to 0 until we find a suitable
   * location
   */
  result = 0;

  /* Begin searching at the start of managed memory */
  current_location = managed_memory_start;

  /* Keep going until we have searched all allocated space */
  while (current_location != last_valid_address) {
    ... look for a free block that is big enough ...
  }
```

Outline di malloc(3) III

```
  /* If we still don't have a valid location, we'll
   * have to ask the operating system for more memory
   */
  if (!result) {
    /* Move the program break numbytes further */
    sbrk(numbytes);

    /* The new memory will be where the last valid
     * address left off
     */
    result = last_valid_address;

    /* We'll move the last valid address forward numbytes */
    last_valid_address = last_valid_address + numbytes;

    /* We need to initialize the mem_control_block */
    current_location_mcb = result;
    current_location_mcb->is_available = 0;
    current_location_mcb->size = numbytes;
  }

  /* Now, no matter what (well, except for error conditions),
   * result has the address of the memory, including
   * the mem_control_block
   */
```

Outline di malloc(3) IV

```
  /* Move the pointer past the mem_control_block */
  result += sizeof(struct mem_control_block);

  /* Return the pointer */
  return result;
}
```

Allocazione di memoria in Linux

Linux usa "demand paging"

Una richiesta di memoria di un processo è considerata "non urgente"

I page frame sono allocati a un processo solo in risposta a un page fault (cioè solo quando è assolutamente indispensabile)

Quando un processo richiede memoria con brk(2) o mmap(2) non ottiene frames; ottiene solo il *diritto* di usare un insieme di indirizzi

Il meccanismo è basato sulle *memory regions*

Memory regions

address space: insieme degli indirizzi che il processo può usare

memory region: un intervallo di indirizzi con determinati *diritti di accesso*

Vengono assegnate nuove regioni a un processo quando:

- il processo "nasce" da una fork(2)
- un processo esegue exec(2)
- un processo esegue mmap(2)
- un processo usa memoria condivisa (shmget(2) ecc.)

Le regioni vengono espanse (o contratte) quando

- lo stack cresce
- il processo esegue brk(2), mmap(2), munmap(2)

Copy-on-write (COW)

fork(2) duplica l'address space di un processo

se il processo è grosso, la copia è molto lenta

prima ottimizzazione: le pagine del *testo* sono condivise perché read-only

seconda ottimizzazione: le pagine dei dati sono condivise in modo *copy-on-write*

quando uno dei processi cerca di modificare una pagina, viene allocato un nuovo frame e la pagina è "sdoppiata"

Implementazione del Copy-on-write in Linux

Durante la fork, tutte le pagine sono marcate read-only

Ma le regioni di dati restano marcate read-write

Quando un processo cerca di modificare una pagina di dati, avviene un page fault

Il S.O. vede che la pagina è *fisicamente* read-only ma *logicamente* read-write

Allora viene allocato un nuovo frame, e le page table di entrambi i processi sono aggiornate

Page fault handling in Linux (overview)