Sistemas de Operação / Fundamentos de Sistemas Operativos

Memory management

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

Outline

Memory management
Introduction

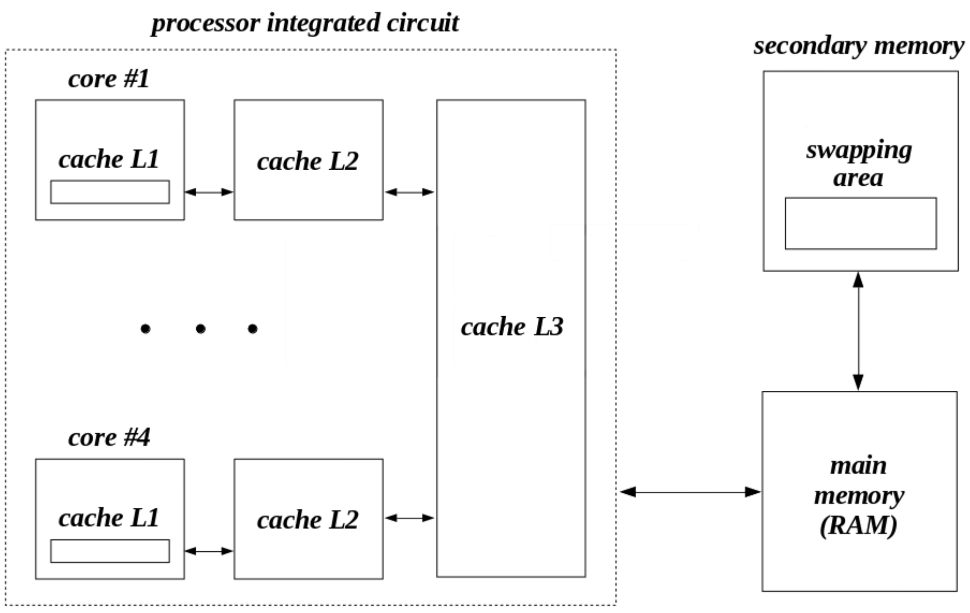
- To be executed, a process must have its address space, at least partially, resident in main memory
- In a multiprogrammed environment, to maximize processor utilization and improve response time (or turnaround time), a computer system must maintain the address spaces of multiple processes resident in main memory
- But, there may not be room for all
  - because, although the main memory has been growing over the years, it is a fact that “data expands to fill the space available for storage”

(Corollary of the Parkinson’s law)

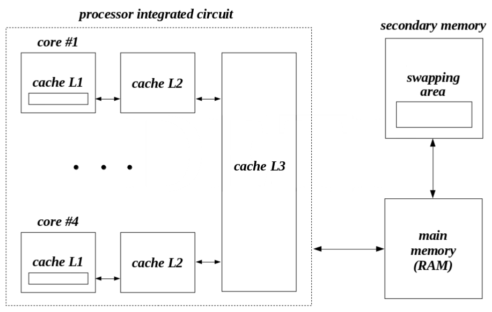
Memory management
Memory hierarchy

- Ideally, an application programmer would like to have infinitely large, infinitely fast, non-volatile and inexpensive available memory
  - In practice, this is not possible
- Thus, the memory of a computer system is typically organized at different levels, forming a hierarchy
  - cache memory – small (tens of KB to some MB), very fast, volatile and expensive
  - main memory – medium size (hundreds of MB to hundreds of GB), volatile and medium price and medium access speed
  - secondary memory – large (tens, hundreds or thousands of GB), slow, non-volatile and cheap

Memory management
Memory hierarchy (2)

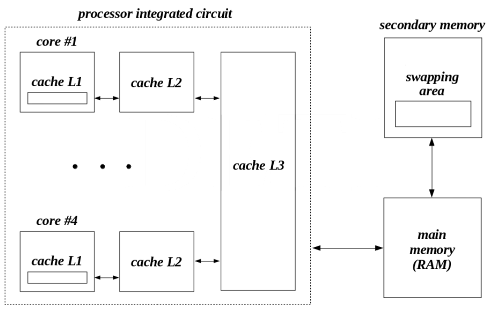


Memory management
Memory hierarchy (3)



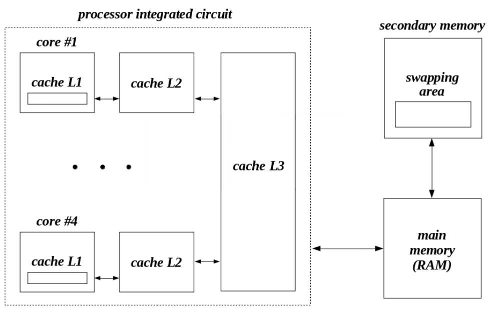
- The cache memory will contain a copy of the memory positions (instructions and operands) most frequently referenced by the processor in the near past
  - The cache memory is located on the processor's own integrated circuit (level 1)
  - And on an autonomous integrated circuit glued to the same substrate (levels 2 and 3)
  - Data transfer to and from main memory is done almost completely transparent to the system programmer

Memory management
Memory hierarchy (3)



- Secondary memory has two main functions
  - File system – storage for more or less permanent information (programs and data)
  - Swapping area – Extension of the main memory so that its size does not constitute a limiting factor to the number of processes that may currently coexist
    - the swapping area can be on a disk partition used only for that purpose or be a file in a file system

Memory management
Memory hierarchy (4)

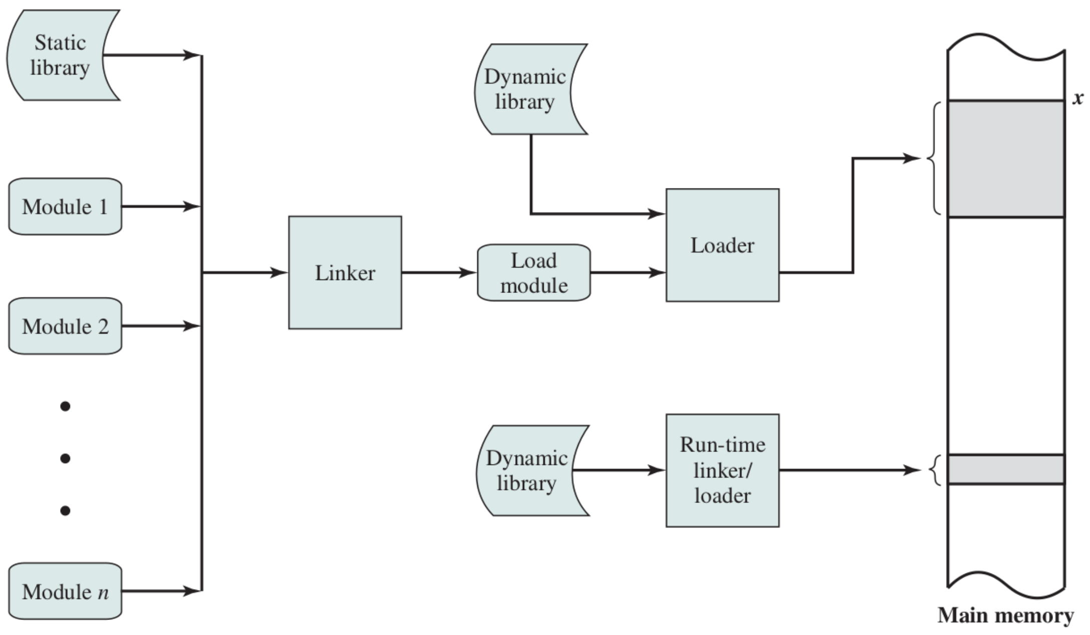


- This type of organization is based on the assumption that the further an instruction or operand is away from the processor, the less times it will be referenced
  - In these conditions, the mean time for a reference tends to be close to the lowest value
- Based on the principle of locality of reference
  - The tendency of a program to access the same set of memory locations repetitively over a short period of time

Memory management
Role

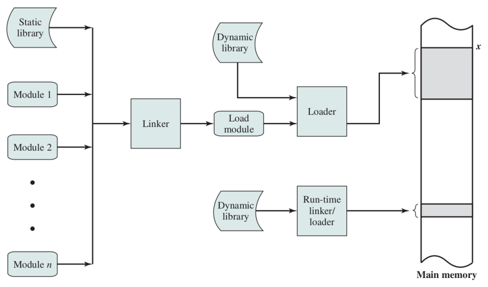
- The role of memory management in a multiprogramming environment focuses on allocating memory to processes and on controlling the transfer of data between main and secondary memory (swapping area), in order to
  - Maintaining a register of the parts of the main memory that are occupied and those that are free
  - Reserving portions of main memory for the processes that will need it, or releasing them when they are no longer needed
  - Swapping out all or part of the address space of a process when the main memory is too small to contain all the processes that coexist.
  - Swapping in all or part of the address space of a process when main memory becomes available

Address space
Linker and loader roles



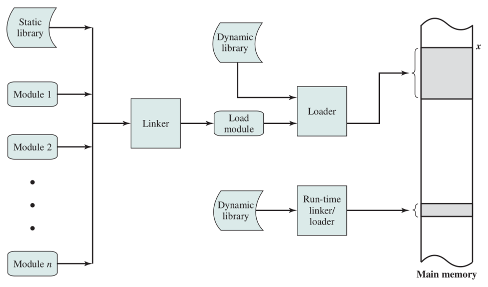
Operating Systems: Internals and Design Principles, William Stallings

Address space
Linker and loader roles (2)



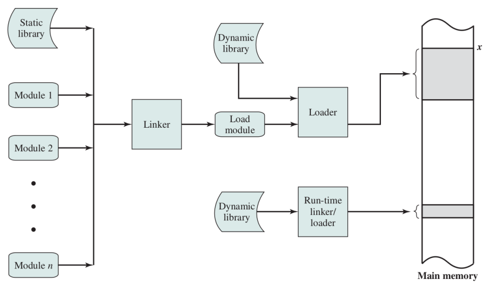
- The object files, resulting from the compilation process, are relocatable files
  - The addresses of the various instructions, constants and variables are calculated from the beginning of the module, by convention the address 0
- The role of the linking process is to bring the different object files together into a single file, the executable file, resolving among themselves the various external references
  - Static libraries are also included in the executable file
  - Dynamic (shared) libraries are not

Address space
Linker and loader roles (3)



- The loader builds the binary image of the process address space, which will eventually be executed, combining the executable file and, if applicable, some dynamic libraries, resolving any remaining external references
- Dynamic libraries can also only be loaded at run time

Address space
Linker and loader roles (4)



- When the linkage is dynamic
  - Each reference in the code to a routine of a dynamic library is replaced by a stub
    - a small set of instructions that determines the location of a specific routine, if it is already resident in main memory, or promotes its load in memory, otherwise
  - When a stub is executed, the associated routine is identified and located in main memory, the stub then replaces the reference to its address in the process code with the address of the system routine and executes it
  - When that code zone is reached again, the system routine is now executed directly
- All processes that use the same dynamic library, execute the same copy of the code, thus minimizing the main memory occupation

Address space
Object and executable files

source file

```
#include    <stdio.h>
#include    <stdlib.h>

int main (void)
{
  printf ("hello, world!\n");
  exit (EXIT_SUCCESS);
}
```

object file

```
$ gcc -Wall -c hello.c

$ file hello.o
hello.o: ELF 64-bit LSB relocatable,
x86-64, version 1 (SYSV), not stripped
```

executable file

```
$ gcc -o hello hello.o
$ file hello
hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=48ac0a8ba08d8df6d5e8a27e00b50248a3061876, not stripped
```

Address space
Object and executable files (2)

```
$ objdump -fstr hello.o
hello.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
SYMBOL TABLE:
0000000000000000 l    df *ABS*          0000000000000000 z.c
0000000000000000 l    d  .text          0000000000000000 .text
0000000000000000 l    d  .data          0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack          0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text          000000000000001a main
0000000000000000         *UND*          0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*          0000000000000000 puts
0000000000000000         *UND*          0000000000000000 exit

...
```

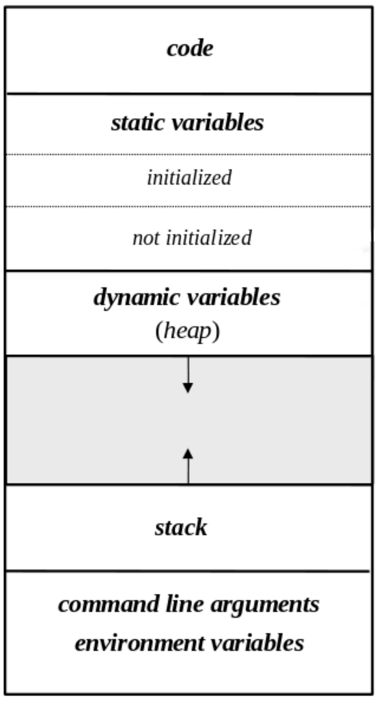
Address space
Object and executable files (3)

```
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000007 R_X86_64_PC32     .rodata-0x0000000000000004
000000000000000c R_X86_64_PLT32    puts-0x0000000000000004
0000000000000016 R_X86_64_PLT32    exit-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text

Contents of section .text:
 0000 554889e5 488d3d00 000000e8 00000000  UH..H.=.........
 0010 bf000000 00e80000 0000               ..........
Contents of section .rodata:
 0000 68656c6c 6f2c2077 6f726c64 2100      hello, world!.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
 0010 332e302d 32377562 756e7475 317e3138  3.0-27ubuntu1~18
 0020 2e303429 20372e33 2e3000             .04) 7.3.0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 1a000000 00410e10 8602430d  .........A....C.
 0030 06000000 00000000                    ........
```

Address space
Object and executable files (4)

```
$ objdump -fTR hello
z:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000000580

DYNAMIC SYMBOL TABLE:
0000000000000000  w   D  *UND*          0000000000000000
_ITM_deregisterTMCloneTable
0000000000000000      DF *UND*          0000000000000000  GLIBC_2.2.5 puts
0000000000000000      DF *UND*          0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000  w   D  *UND*          0000000000000000              __gmon_start__
0000000000000000      DF *UND*          0000000000000000  GLIBC_2.2.5 exit
0000000000000000  w   D  *UND*          0000000000000000
_ITM_registerTMCloneTable
0000000000000000  w   DF *UND*          0000000000000000  GLIBC_2.2.5 __cxa_finalize

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000200db0 R_X86_64_RELATIVE  *ABS*+0x0000000000000680
0000000000200db8 R_X86_64_RELATIVE  *ABS*+0x0000000000000640
0000000000201008 R_X86_64_RELATIVE  *ABS*+0x0000000000201008
0000000000200fd8 R_X86_64_GLOB_DAT  _ITM_deregisterTMCloneTable
0000000000200fe0 R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.2.5
0000000000200fe8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000200ff0 R_X86_64_GLOB_DAT  _ITM_registerTMCloneTable
0000000000200ff8 R_X86_64_GLOB_DAT  __cxa_finalize@GLIBC_2.2.5
0000000000200fc8 R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
0000000000200fd0 R_X86_64_JUMP_SLOT  exit@GLIBC_2.2.5
```

Address space
Address space of a process

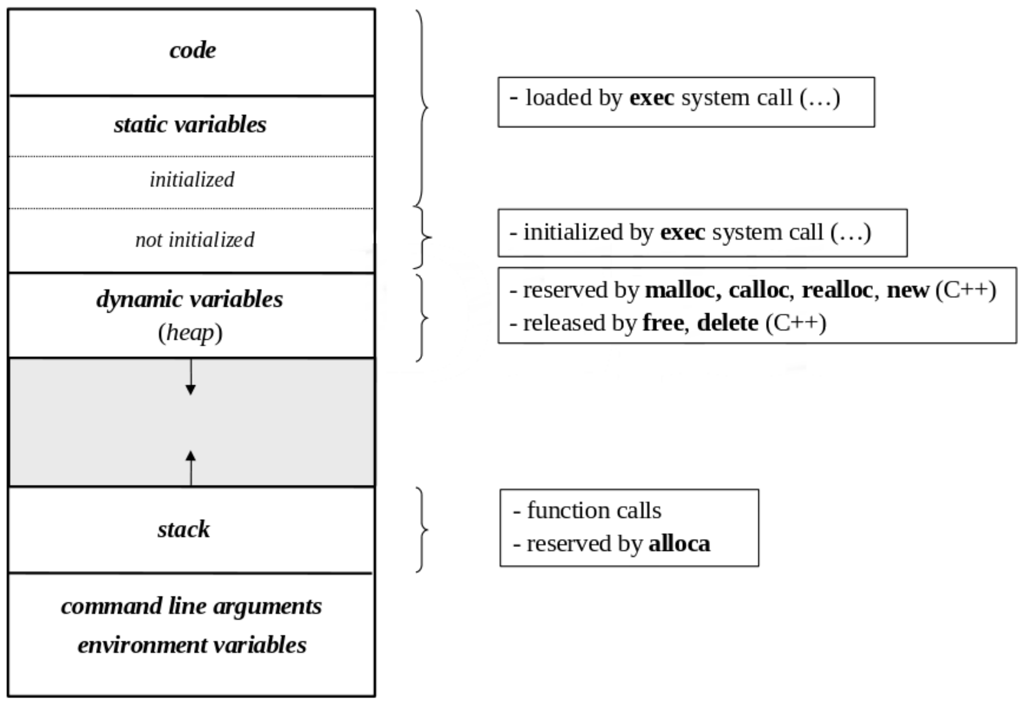


- Code and static variables regions have a fixed size, which is determined by the loader
- Dynamic variables and stack regions grow (in opposite directions) during the execution of the process
- It is a common practice to leave an unallocated memory area in the process address space between the dynamic definition region and the stack that can be used alternatively by any of them
- When this area is exhausted on the stack side, the execution of the process cannot continue, resulting in the occurrence of a fatal error: stack overflow

Address space
Address space of a process

- The binary image of the process address space represents a relocatable address space, the so-called logical address space
- The main memory region where it is loaded for execution, constitutes the physical address space of the process
- Separation between the logical and physical address spaces is a central concept to the memory management mechanisms in a multiprogrammed environment
- There are two issues that have to be solved
  - dynamic mapping – ability to convert a logical address to a physical address at runtime, so that the physical address space of a process can be placed in any region of main memory and be moved if necessary
  - dynamic protection – ability to prevent at runtime access to addresses located outside the process's own address space

Logical address space
Overview

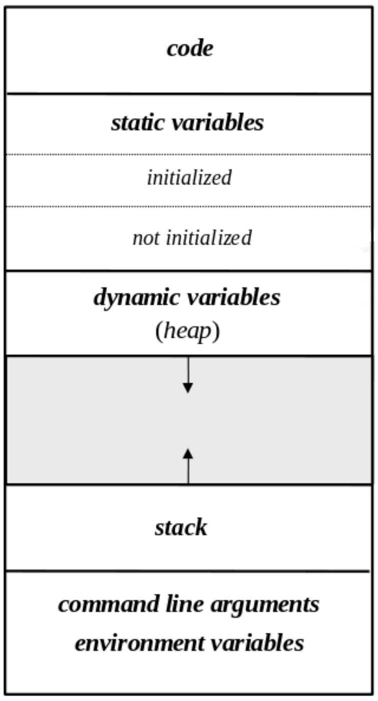

code
static variables
initialized
not initialized
dynamic variables
(heap)
stack
command line arguments
environment variables
- loaded by **exec** system call (…)
- initialized by **exec** system call (…)
- reserved by **malloc, calloc, realloc, new** (C++)
- released by **free, delete** (C++)
- function calls
- reserved by **alloca**

Logical address space
Command line arguments and environment variables

```c
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>

int main(int argc, char *argv[], char *env[])
{
    /* printing command line arguments */
    printf("Command line arguments:\n");
    for (int i = 0; argv[i] != NULL; i++)
    {
        printf("  %s\n", argv[i]);
    }

    /* printing all environment variables */
    printf("\nEnvironment variables:\n");
    for (int i = 0; env[i] != NULL; i++)
    {
        printf("  %s\n", env[i]);
    }

    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf("  env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf("  env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

    return EXIT_SUCCESS;
}
```

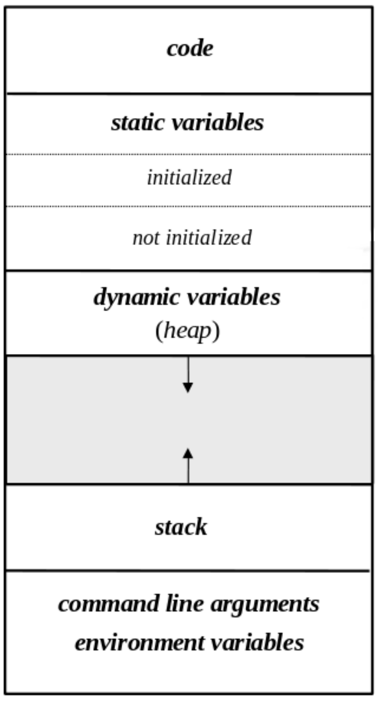
- argv is an array of strings
- argv[0] is the program reference
- env is an array of strings, each representing a variable, in the form name-value pair
- getenv returns the value of a variable name

Logical address space
Logical addresses of variables



```c
// n0 is defined in the environment
int n1 = 1;              // global, initialized
static int n2 = 2;       // static, file-scoped, initialized
int n3;                  // global, not initialized
static int n4;           // static, file-scoped, not initialized
int n5;                  // another global, not initialized
static int n6 = 6;       // another static, file-scoped, initialized

int main(int argc, char *argv[], char *env[])
{
    extern char** environ;
    static int n7;       // static, function-scoped, not initialized
    static int n8 = 8;   // static, function-scoped, initialized
    int *p9 = (int*)malloc(sizeof(int));    // heap-dynamic
    int *p10 = new int;                     // heap-dynamic
    int *p11 = (int*)alloca(sizeof(int));   // stack-dynamic
    int n12;                                // local, not initialized
    int n13 = 13;                           // local, initialized
    int n14;                                // local, not initialized

    printf("\ngetenv(n0): %p\n", getenv("n0"));
    printf("\nargv: %p\nenviron: %p\nenv: %p\nmain: %p\n\n",
           argv, environ, env, main);
    printf("\n&argc: %p\n&argv: %p\n&env: %p\n",
           &argc, &argv, &env);
    printf("&n1: %p\n&n2: %p\n&n3: %p\n&n4: %p\n&n5: %p\n"
           "&n6: %p\n&n7: %p\n&n8: %p\nn9: %p\nn10: %p\n"
           "p11: %p\n&n12: %p\n&n13: %p\n&n14: %p\n",
           &n1, &n2, &n3, &n4, &n5, &n6, &n7, &n8,
           p9, p10, p11, &n12, &n13, &n14);
```

Logical address space
Logical address space after a `fork`



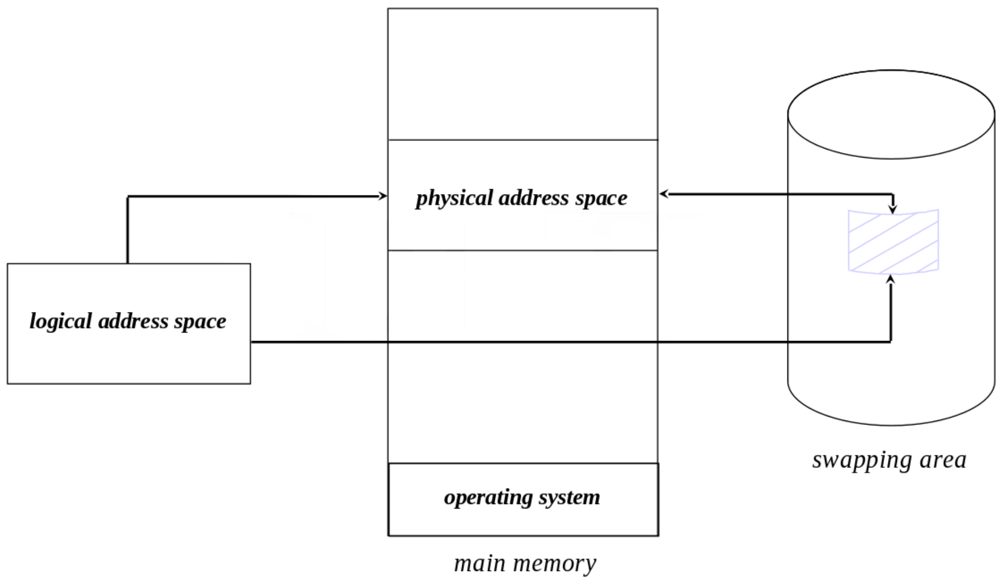
```
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <wait.h>

int n01 = 1;

int main(int argc, char *argv[], char *env[])
{
    int pid = fork();
    if (pid != 0)
    {
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
        wait(NULL);
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
    }
    else
    {
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
        n01 = 1111;
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
    }
    return 0;
}
```

Logical address space
Logical addresses between `threads`

```
void *threadChild (void *par)
{
  printf ("I'm the child thread! (PID: %d; TID: %d)\n", getpid(), gettid());

  int n1 = 0;
  static int n2 = 0;
  printf("[%u] &n1: %p; &n2: %p\n", gettid(), &n1, &n2);

  return NULL;
}

int main (int argc, char *argv[])
{
  printf ("I'm the main thread! (PID: %d)\n", getpid());

  threadChild(NULL);
  threadChild(NULL);

  pthread_t thr[2];
  for (int i = 0; i < 2; i++) {
      if (pthread_create (&thr[i], NULL, threadChild, NULL) != 0) {
        perror ("Fail launching thread");
        return EXIT_FAILURE;
      }
  }

  for (int i = 0; i < 2; i++) {
      if (pthread_join (thr[i], NULL) != 0) {
          perror ("Fail joining child thread");
          return EXIT_FAILURE;
      }
  }

  threadChild(NULL);

  return EXIT_SUCCESS;
}
```

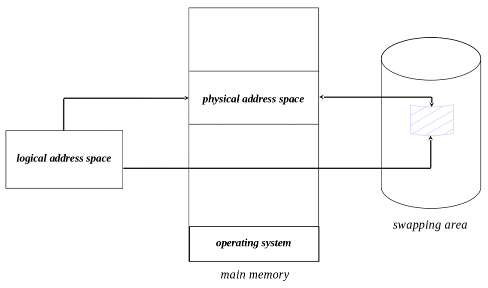
Contiguous memory allocation
Logical and physical address spaces

- In contiguous memory allocation, there is a one-to-one correspondence between the logical address space of a process and its physical address space



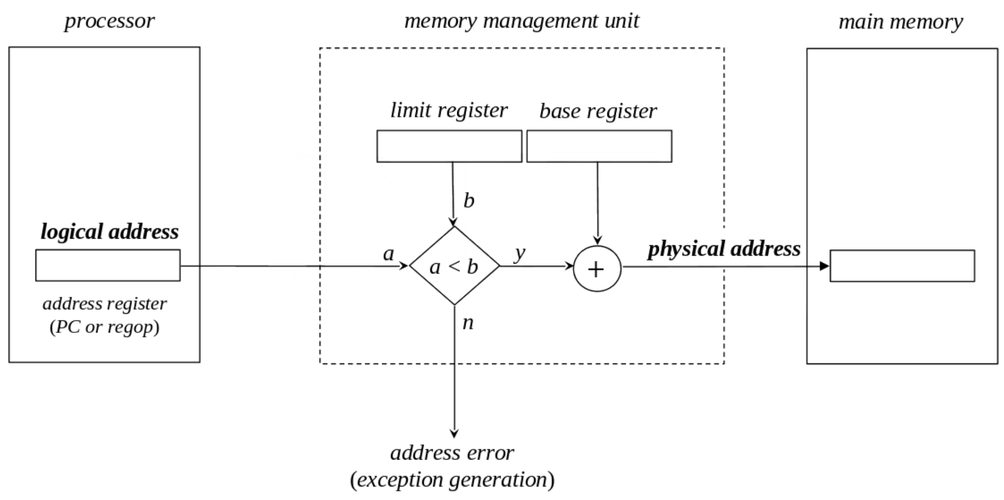
Contiguous memory allocation
Logical and physical address spaces

- Consequences:
  - Limitation of the address space of a process – in no case can memory management support automatic mechanisms that allow the address space of a process to be larger than the size of the main memory available
    - The use of overlays can allow to overcome that
  - Contiguity of the physical address space – although it is not a strictly necessary condition, it is naturally simpler and more efficient to assume that the process address space is contiguous
  - Swapping area as an extension of the main memory – it serves to storage the address space of processes that cannot be resident into main memory due to lack of space

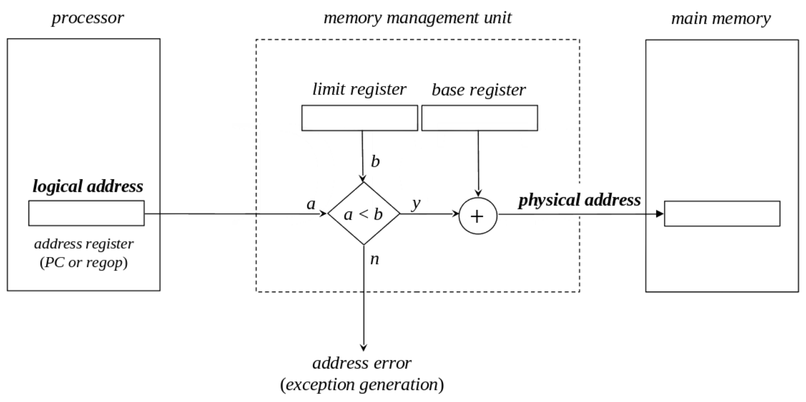


Contiguous memory allocation
Logical address to physical address translation

- How are dynamic mapping and dynamic protection accomplished?
  - A piece of hardware (the MMU) comes into play

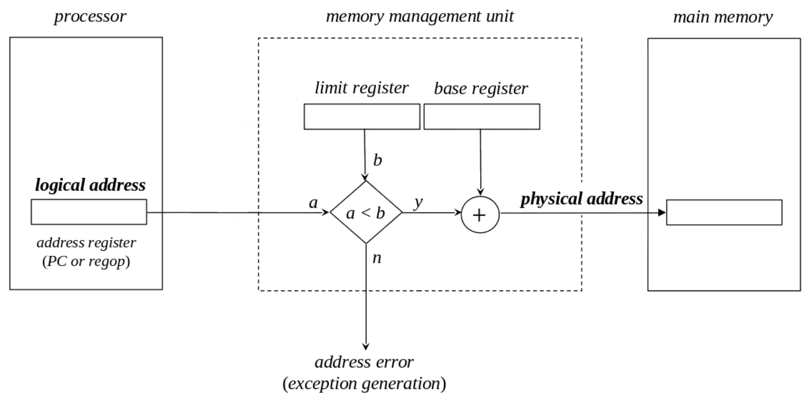


Contiguous memory allocation
Logical address to physical address translation (2)



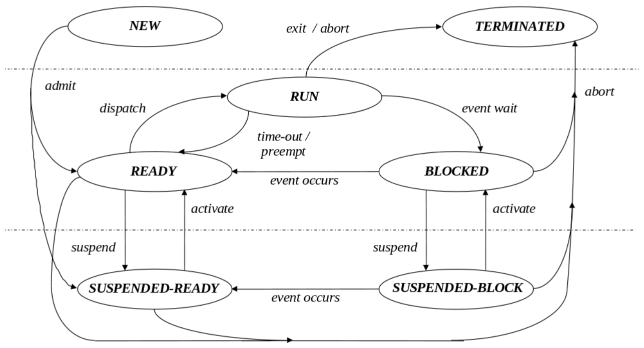
- The limit register must contain the size in bytes of the logical address space
- The base register must contain the address of the beginning of the main memory region where the physical address space of the process is placed
- On context switching, the dispatch operation loads the base and limit registers with the values present in the corresponding fields of the process control table entry associated with the process that is being scheduled for execution

Contiguous memory allocation
Logical address to physical address translation (2)



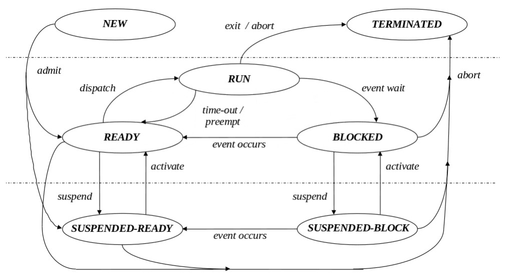
- Whenever there is a reference to memory
  - the logical address is first compared to the value of the limit register
  - if it is greater than or equal to, it is an invalid reference, a null memory access (dummy cycle) is set in motion and an exception is generated due to address error
  - otherwise, it is a valid reference (it occurs within the process address space), the logical address is added to the value of the base register to produce the physical address

Contiguous memory allocation
Long-term scheduling



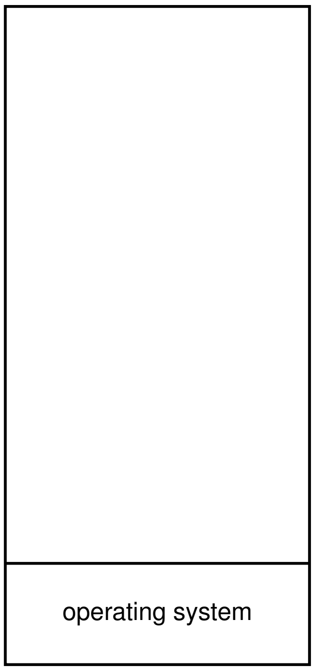
- When a process is created, the data structures to manage it is initialized
  - Its logical address space is constructed, and the value of the limit register is computed and saved in the corresponding field of the process control table (PCT)
- If there is space in main memory, its address space is loaded there, the base register field is updated with the initial address of the assigned region and the process is placed in the READY queue
- Otherwise, its address space is temporarily stored in the swapping area and the process is placed in the SUSPENDED-READY queue

Contiguous memory allocation
Medium-term scheduling



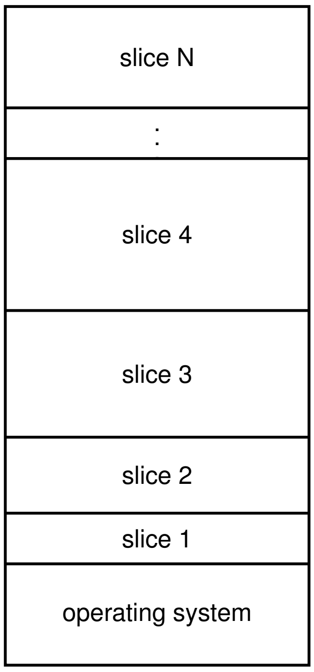
- If memory is required for another process, a BLOCKED (or even READY) process may be swapped out, freeing the physical memory it is using,
  - In such a case, its base register field in the PCT becomes undefined
- If memory becomes available, a SUSPENDED-READY (or even SUSPENDED-BLOCKED) process may be swapped in,
  - Its base register field in the PCT is updated with its new physical location
  - A SUSPENDED-BLOCK process is only selected if no SUSPENDED-READY one exists
- When a process terminates, it is swapped out (if not already there), waiting for the end of operations

Memory partitioning
How to do it?



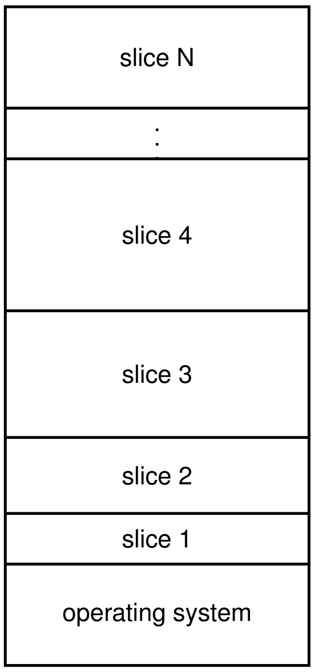
- After reserving some amount to the operating system, how to partition the real memory to accommodate the different processes?
  - Fixed-size partitioning
    - into slices of equal size
    - into slices of different size
  - Dynamic partitioning
    - being done as being requested

Memory partitioning
Fixed-size partitioning



- Main memory can be divided into a number of static slices at system generation time
  - not necessarily all the same size
- The logical address space of a process may be loaded into a slice of equal or greater size
  - thus, the largest slice determines the size of the largest allowable process
- Some features:
  - Simple to implement
  - Efficient – little operating system overhead
  - Fixed number of allowable processes
  - Inefficient use of memory due to internal fragmentation – the part of a slice not used by a process is wasted

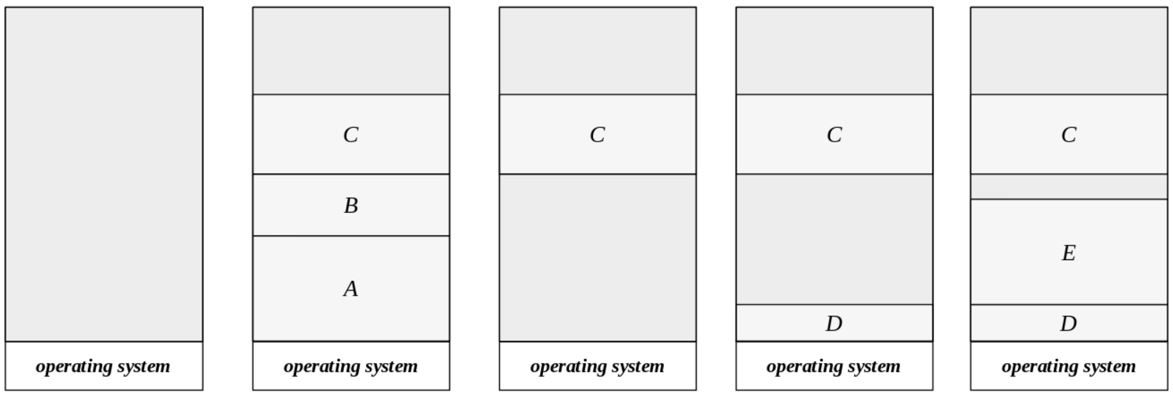
Memory partitioning
Fixed-size partitioning (2)



- If a slice becomes available, which of the SUSPENDED-READY processes should be placed there?
- Two different scheduling policies are here considered
  - Valuing fairness – the first process in the queue of SUSPENDED-READY processes whose address space fits in the slice is chosen
  - Valuing the occupation of main memory – the first process in the queue of SUSPENDED-READY processes with the largest address space that fits in the slice is chosen
    - to avoid starvation an aging mechanism can be used

Memory partitioning
Dynamic partitioning

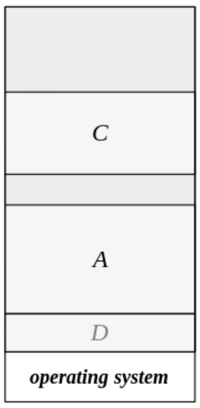
- In dynamic partitioning, at start, all the available part of the memory constitutes a single block and then
  - reserve a region of sufficient size to load the address space of the processes that arises
  - release that region when it is no longer needed

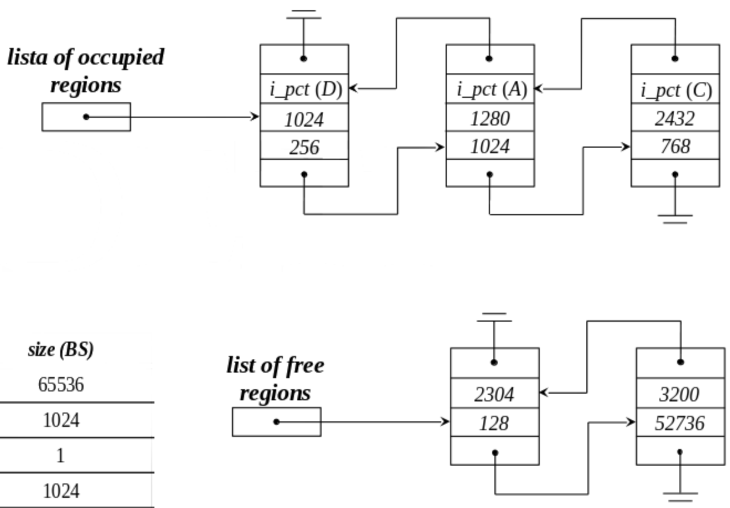


Memory partitioning
Dynamic partitioning (2)

- As the memory is dynamically reserved and released, the operating system has to keep an updated record of occupied and free regions
- One way to do this is by building two (bi)linked lists
  - list of occupied regions – locates the regions that have been reserved for storage of the address spaces of processes resident in main memory
  - list of free regions – locates the regions still available
- Memory is not allocated in byte boundaries, because
  - useless, very small free regions may appear
  - that will be included in the list of free regions
  - making subsequent searches more complex
- Thus, the main memory is typically divided into blocks of fixed size and allocation is made in units of these blocks

Memory partitioning
Dynamic partitioning (3)





	size (bytes)	size (BS)
main memory	256 M	65536
operating system	4M	1024
block size (BS)	4K	1
process A	4M	1024
Process C	3M	768
Process D	1M	256

Memory partitioning
Dynamic partitioning (4)

- Valuing fairness is the scheduling discipline generally adopted, being chosen the first process in the queue of SUSPENDED-READY processes whose address space can be placed in main memory
- Dynamic partitioning can produce external fragmentation
  - Free space is splitted in a large number of (possible) small free regions
  - Situations can be reached where, although there is enough free memory, it is not continuous and the storage of the address space of a new or suspended process is no longer possible
- The solution is garbage collection – compact the free space, grouping all the free regions into a single one
  - This operation requires stopping all processing and, if the memory is large, can have a very long execution time

Memory partitioning
Dynamic partitioning (5)

- In case there are several free regions available, which one to use to allocate the address space of a process?
- Possible policies:
  - first fit – the list of free regions is searched from the beginning until the first region with sufficient size is found
  - next fit – is a variant of the first fit which consists of starting the search from the stop point in the previous search
  - best fit – the list of free regions is fully searched, choosing the smallest region with sufficient size for the process
  - worst fit – the list of free regions is fully searched, choosing the largest existing region
  - buddy system – which uses a binary tree to represent used or unused memory blocks and splits memory into halves to try to give a best fit
- Which one is the best?
  - in terms of fragmentation
  - in terms of efficiency of allocation
  - in terms of efficiency of release

Memory partitioning
Dynamic partitioning (6)

- Advantages
  - general – the scope of application is independent of the type of processes that will be executed
  - low complexity implementation – no special hardware required and data structures are reduced to two (bi)linked lists
- Disadvantages
  - external fragmentation – the fraction of the main memory that ends up being wasted, given the small size of the regions in which it is divided, can reach in some cases about a third of the total (50% rule)
  - inefficient – it is not possible to build algorithms that are simultaneously very efficient in allocating and freeing space