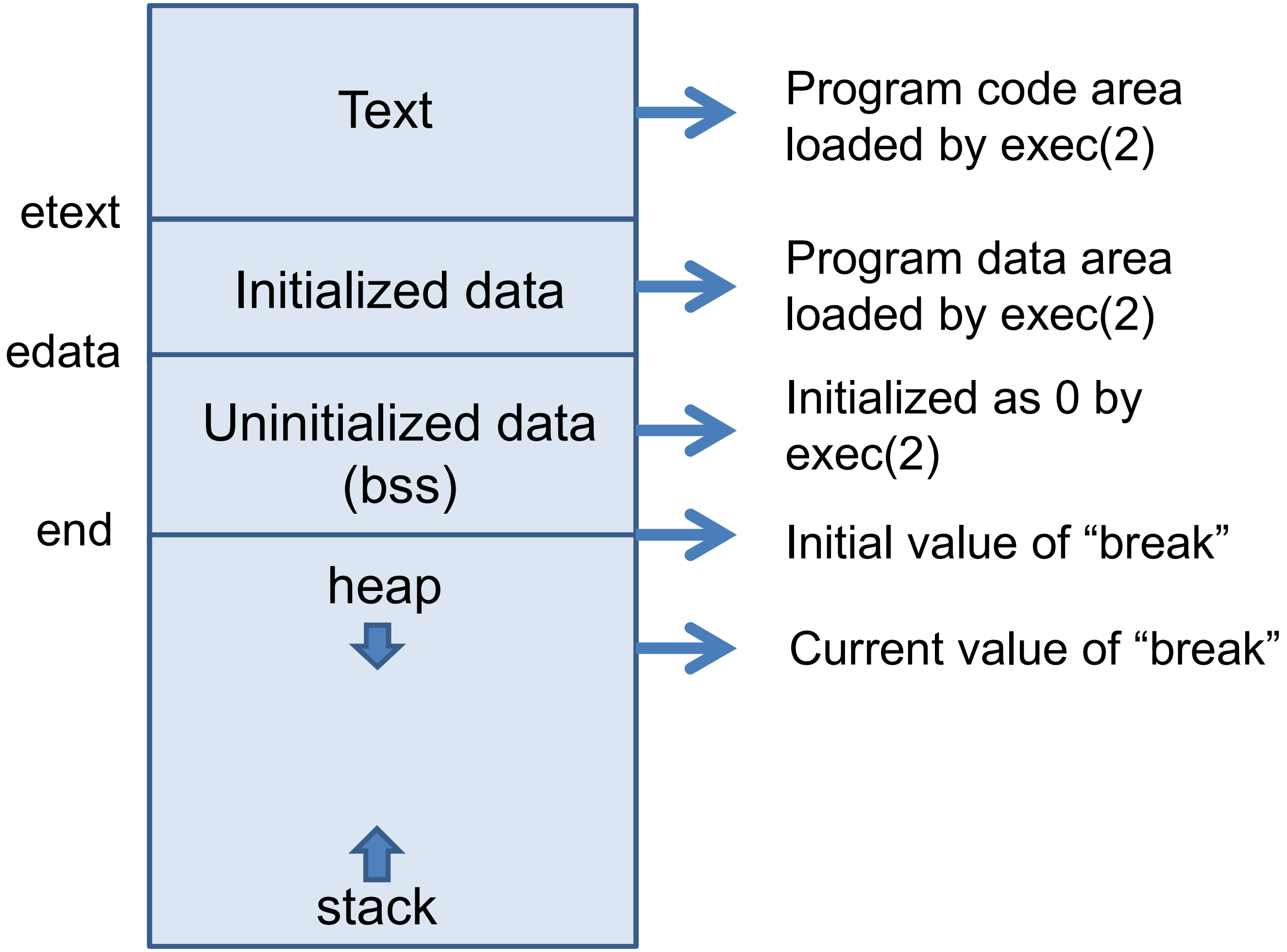
System Programming

9. Memory

Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Process Address Space



Program code area loaded by exec(2)

Program data area loaded by exec(2)

Initialized as 0 by exec(2)

Initial value of “break”

Current value of “break”

Process Address Space

- **Text** Area
  - CPU instructions of a program
  - Read-only constants
- **Data** Area
  - Global initialized data area + global uninitialized data area (***bss = block started by symbol***)
- **Heap** Area
  - For dynamic memory allocation (malloc(), new, etc.)
- **Stack** Area
  - automatic variables in a function
  - Call frames (arguments, return address, etc.)

Memory for main()

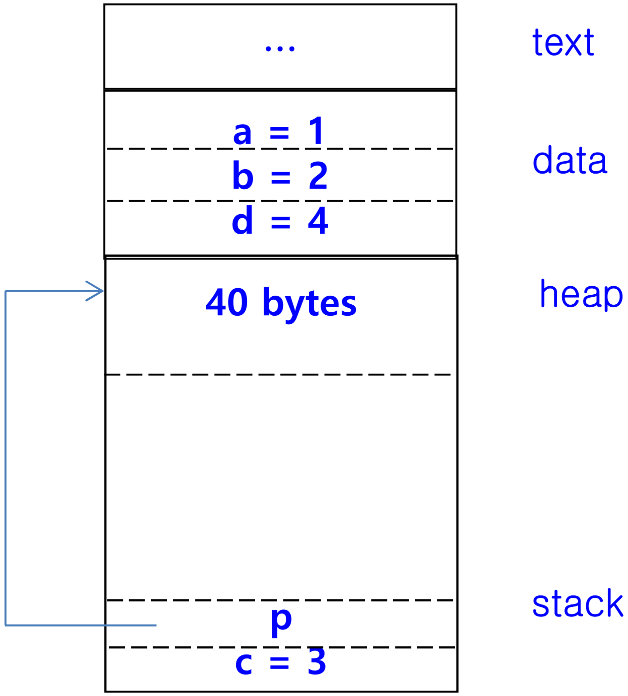
- vars.c

```
#include <stdio.h>
#include <stdlib.h>
int a = 1;
static int b = 2;

int main() {
    int c = 3;
    static int d = 4;
    char *p;

    p = (char *) malloc(40);
    fun(5);
}

void fun(int n)
{
    int m = 6;
     ...
}
```



Memory for fun()

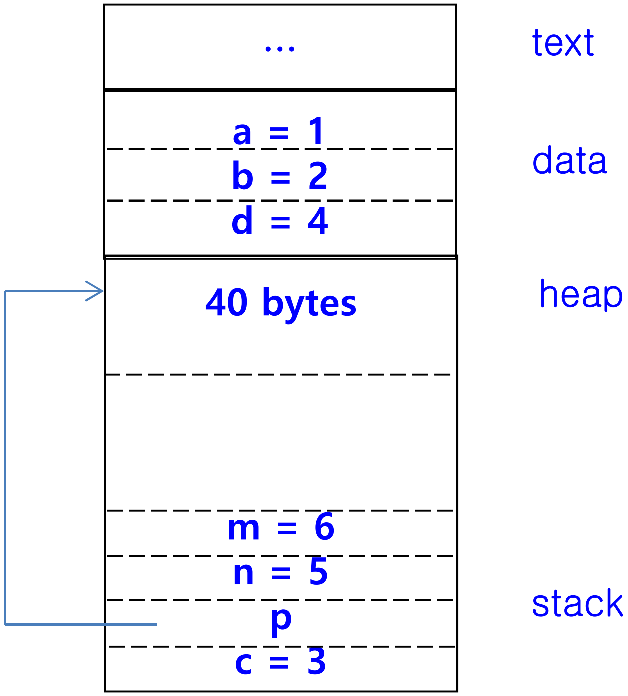
- vars.c

```
#include <stdio.h>
#include <stdlib.h>
int a = 1;
static int b = 2;

int main() {
    int c = 3;
    static int d = 4;
    char *p;

    p = (char *) malloc(40);
    fun(5);
}

void fun(int n)
{
    int m = 6;
     ...
}
```



Dynamic Memory Allocation

- Reasons to use dynamic allocation
  - This saves memory by requesting and using as much memory as needed when needed.

- malloc( )
- calloc( )
- realloc( )

- free( )

Memory Allocation

```
#include <stdlib.h>
void *malloc(size_t size);

void free(void *ptr);
```

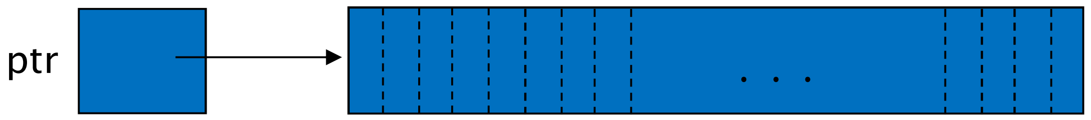
- Allocate memory space of size bytes and return the starting address in void * type.

- parameters:
  - *size* : Allocate memory space of size bytes

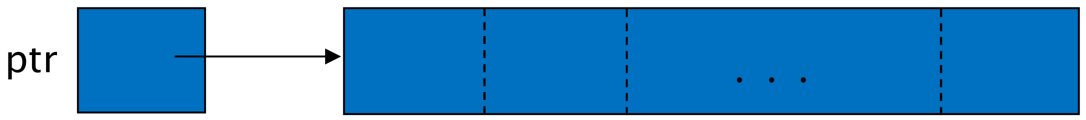
- return:
  - return the starting address in void * type

Memory Allocation usage

```
char *ptr;
ptr = (char *) malloc(40);
```

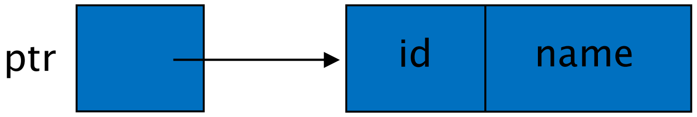


```
int *ptr;
ptr = (int *) malloc(10 * sizeof(int));
```

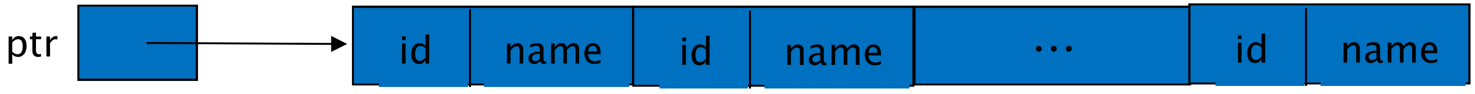


Memory Allocation usage

```
struct student {
        int id;
        char name[10];
};
struct student *ptr;
ptr = (struct student *) malloc(sizeof(struct student));
```



```
struct student *ptr;
ptr = (struct student *) malloc(n * sizeof(struct student));
```



Memory allocation ex1

- stud1.c

```
#include <stdio.h>
#include <stdlib.h>
struct student {
    int id;
    char name[20];
};

int main()
{
    struct student *ptr;
    int n, i;
    printf(“How many student ? ");
    scanf("%d", &n);
    if (n <= 0) {
        fprintf(stderr, “errpr: wrong number.\n");
        fprintf(stderr, “terminate program\n");
        exit(1);
    }
```

Memory allocation ex1

- stud1.c

```
 ptr = (struct student *) malloc(n * sizeof(struct student));
 if (ptr == NULL) {
     perror("malloc");
        exit(2);
 }

 printf(“enter student number and name for %d. students\n", n);
 for (i = 0; i < n; i++)
     scanf("%d %s\n", &ptr[i].id, ptr[i].name);

 printf("\n* student information *\n");
 for (i = n-1; i >= 0; i--)
    printf("%d %s\n", ptr[i].id, ptr[i].name);

 printf("\n");
 exit(0);
}
```

Memory Allocation

```
#include <stdlib.h>
void *calloc(size_t n, size_t size);

#include <stdlib.h>
void *realloc(void *ptr, size_t newsize);
```

- calloc : Allocate n memory spaces of size size. Initialize all values to zero. On failure, NULL is returned.
- Parameter
  - n : number of spaces to be allocated
  - size : size of each space
- realloc : Change the size of the memory allocated by ptr to newsize.
- Parameters
  - ptr : pointer for the allocated memory
  - newsize : size for new allocation

Memory Mapped File

- Objectives
  - By mapping a memory area (e.g. *struct*) into a file,
    - ***Auto-saving*** of memory variables into a file at run time.
    - Memory-reading = reading form a file
    - Memory-writing = writing to a file (to ***page cache***)
    - Handle a file data as an memory array or using a pointer var.
  - No copy b/w library buffer and kernel buffer

    → improves the file I/O performance
  - When several processes map their memory areas into the same file,
    - ***Inter-process communication is possible*** by reading and writing memory area

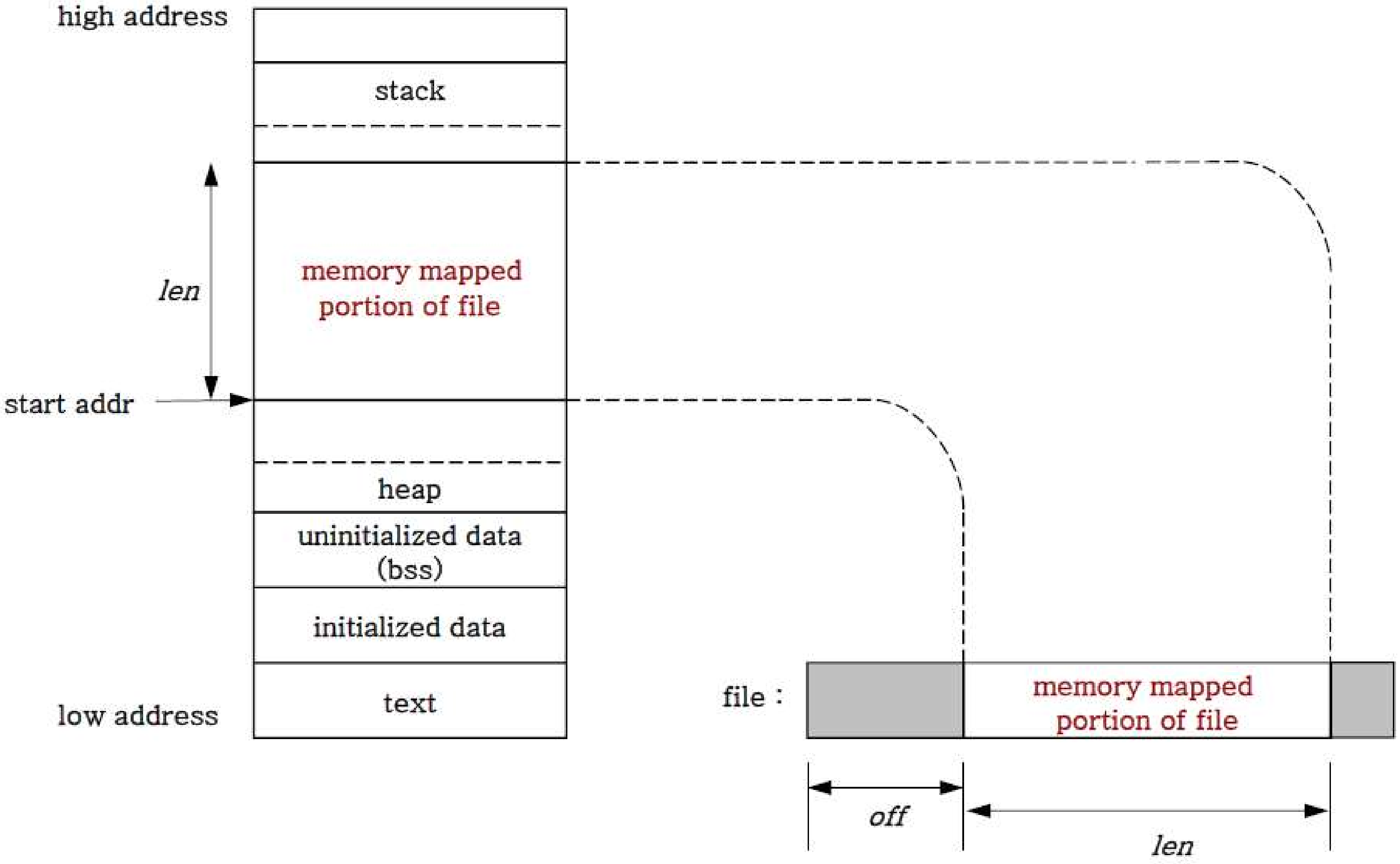
Memory Mapped File

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap (caddr_t addr, size_t len, int prot, int flags,
int fd, off_t offset);
```

- parameters:
  - *addr* : starting address of a memory area to be mapped to a file
  - *len* : mapping size
  - *prot* : access permission to the memory area
  - *flags* : the mapping scheme
  - *fd* : the file descriptor to be mapped
  - *offset* : the offset in a file where the mapping starts
- return:
  - the real starting address of a memory area to be mapped
    - the kernel may map a file into a different area
  - error : MAP_FAILED

File-memory Mapping by *mmap( )*



prot and **flags** arguments

prot	meaning
PROT_READ	Reading is possible
PROT_WRITE	Writing is possible
PROT_EXEC	Execution is possible
PROT_NONE	Nothing is allowed
* the protection specified for a region has to match the `open` mode of the file.	

flags	meaning
MAP_SHARED*	Writing to memory is synchronized to the file
MAP_PRIVATE*	Writing is occurred only in another file copy, and the original file is not modified
MAP_FIXED	Resulting map address must be same as the address given as an argument. If not MAP_FIXED, the address argument is just for reference.
MAP_NORESERVE	Do not preserve a swap space
* either MAP_SHARED or MAP_PRIVATE should be specified.	

Release a memory mapping

```
#include <sys/types.h>
#include <sys/mman.h>

int munmap (caddr_t addr, size_t len);
```

- parameters:
  - *addr* : the starting address of a memory area to be unmapped
  - *len* : length
- return:
  - 0 if OK
  - -1 on an error

File copy using mmap file (1)

mmcp.c

```
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <fcntl.h>

#define FILE_MODE  (S_IRUSR | S_IWUSR)

void mperr(char *call, int val)
{
    perror( call);
    exit( val);
}

int main( int argc, char *argv[])
{
    int fdin, fdout;
    caddr_t src, dst;
    struct stat statbuf;
```

File copy using mmap file (2)

```
if (argc != 3)
        mperr ("usage : a.out <fromfile> <tofile>", 1);

if ((fdin = open ( argv[1], O_RDONLY)) < 0) {
        fprintf(stderr, "cannot open %s for writing", argv[1]);
        exit(2);
}
if ((fdout = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0) {
        fprintf(stderr, "cannot create %s for writing", argv[2]);
        exit(3);
}

if (fstat(fdin, &statbuf ) < 0)
        mperr("fstat error", 4);

if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        mperr("lseek error", 5);

if (write(fdout, "", 1) != 1)
        mperr("write error", 6);
```

File copy using mmap file (4)

```
        if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED, fdin, 0)) == MAP_FAILED)
                mperr("mmap error for input", 7);

        if ((dst = mmap(0, statbuf.st_size, PROT_WRITE, MAP_SHARED, fdout, 0)) == MAP_FAILED)
                mperr("mmap error for output", 8);

        memcpy(dst, src, statbuf.st_size);


        if(munmap(src, statbuf.st_size) != 0)
                mperr("munmap(src) error", 9);
        if(munmap(dst, statbuf.st_size) != 0)
                mperr("munmap(src) error", 10);

        exit(0);
}
```

Run & Result

```
$ cat myfile
This is a test data.
$ ./a.out myfile mycopy
$ ls -ld myfile mycopy
-rw-rw-r--   1   shlim  shlim    512 Jul 9 22 : 43 myfile
-rw-rw-r--   1   shlim  shlim    512 Jul 9 24 : 13 mycopy
$ cat mycopy
This is a test data.
$
```

Disk Synchronization of a MM Area

```
#include <sys/mman.h>
#include <unistd.h>

int msync (const void *addr, size_t len, int flags);
```

- parameters:
  - *addr* : starting address
  - *len* : length
  - *flags* : control **msync( )** operation
- return:
  - 0 if OK
  - -1 on an error

flags argument

flags	meaning
MS_ASYNC	Asynchronous sync request. Do not wait the completion of the sync.
MS_SYNC	Synchronous sync request. Be blocked(waits) until the sync has been completed
MS_INVALIDATE	Invalidate other memory mapping on the same file

msync ex

msync-ex.c

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MEM_SIZE 64

int main(int argc, char **argv)
{
    int fd;
    char *memPtr = NULL;
    struct stat sb;
    int flag = PROT_WRITE | PROT_READ;

    if (argc != 2)    {
        fprintf(stderr, "Usage: %s memPtr\n", argv[0]);
        exit(1);
    }
```

msync ex

msync-ex.c

```
if ((fd = open(argv[1], O_RDWR | O_CREAT)) < 0){
    perror("File Open Error");
    exit(1);
}

if (fstat(fd, &sb) < 0)     {
    perror("fstat error");
    exit(1);
}

memPtr = (char *)mmap(0, MEM_SIZE, flag, MAP_SHARED, fd, 0);
if (memPtr == (void *)-1) {
   perror("mmap() error");
   close(fd);
   exit(1);
}

printf("mem(%p),value(%s)\n", memPtr, memPtr);
```

msync ex

msync-ex.c

```c
    // mem <--> file (synchronization)
    while(1){
          scanf("%s", memPtr);
          if (!strcmp(memPtr, "exit")) break;
          if (msync(memPtr, MEM_SIZE, MS_SYNC) == -1)
          {
                printf("mync() error(%s)\n", strerror(errno));
                break;
          }
    }

    if (munmap(memPtr, MEM_SIZE) == -1) {
          printf("munmap() error(%s)\n", strerror(errno));
    }
    close(fd);
}
```