Heap

2019 Spring

Lecture Outline

- Heap-allocated Memory
  - `malloc()` and `free()`
  - Memory leaks
  - Sample codes

Memory Allocation So Far (1/2)

- So far, we have seen two kinds of memory allocation:

```
int counter = 0;      // global var

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

Memory Allocation So Far (2/2)

- So far, we have seen two kinds of memory allocation:

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than can fit on the Stack
  - We need memory whose size is not known in advance to the caller

```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)
- C requires you to manually manage memory
  - Gives you more control, but causes headaches

Aside: `NULL`

- `NULL` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

segfault.c

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return 0;
}
```

`malloc()`

- General usage:

```
var = (type*) malloc(size in bytes)
```

- `malloc` allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns `NULL` if the memory allocation failed!
  - You should assume that the memory initially contains ga rbage
  - You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

`calloc()`

- General usage:

`var = (type*) calloc(`*num*, *bytes per element*`)`

- Like `malloc`, but also zeros out the block of memory
  - Helpful for shaking out bugs
  - Slightly slower; preferred for non-performance-critical code
  - `malloc` and `calloc` are found in `stdlib.h`

```
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

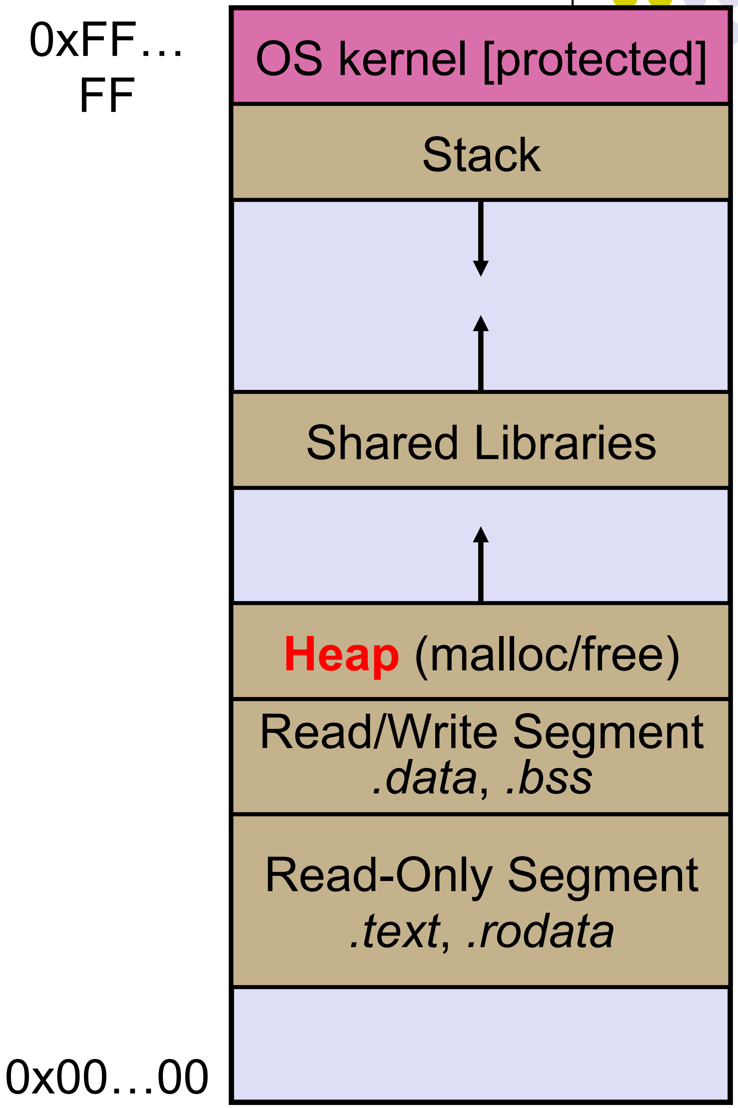
`free()`

- Usage: `free(pointer);`
- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by `malloc` or `calloc`)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to free
    - Defensive programming: can set pointer to `NULL` after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...              // do stuff with arr
free(arr);
arr = NULL;    // OPTIONAL
```

The Heap

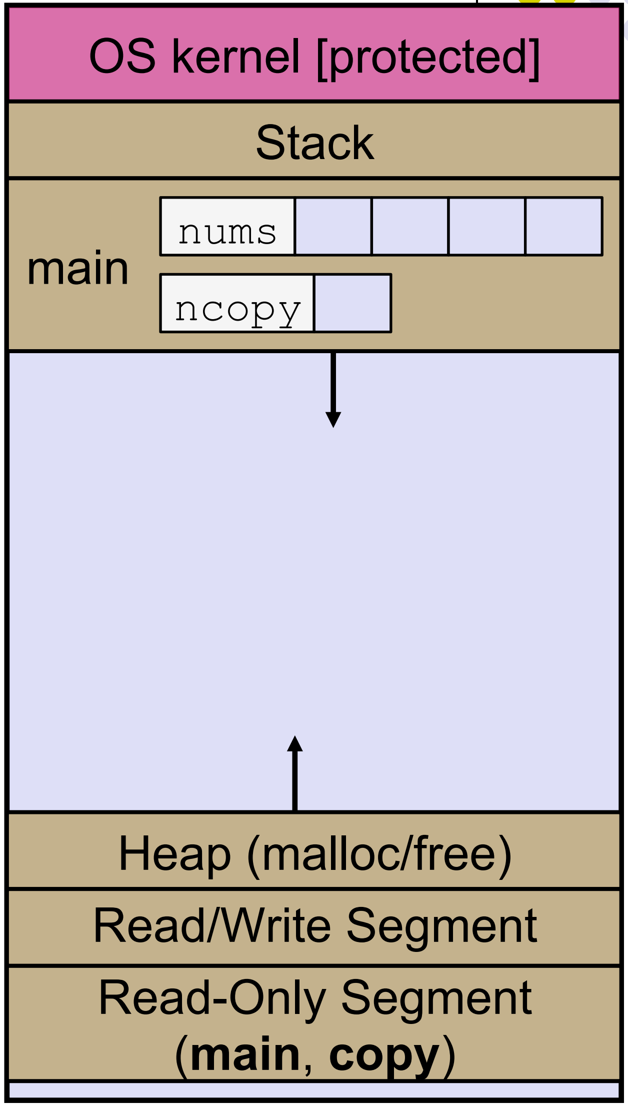
- The Heap is a large pool of unused memory that is used for dynamically-allocated data
  - **`malloc`** allocates chunks of data in the Heap; **`free`** deallocates those chunks
  - **`malloc`** maintains bookkeeping data in the Heap to track allocated blocks



Heap and Stack Example (1/11)

arraycopy.c

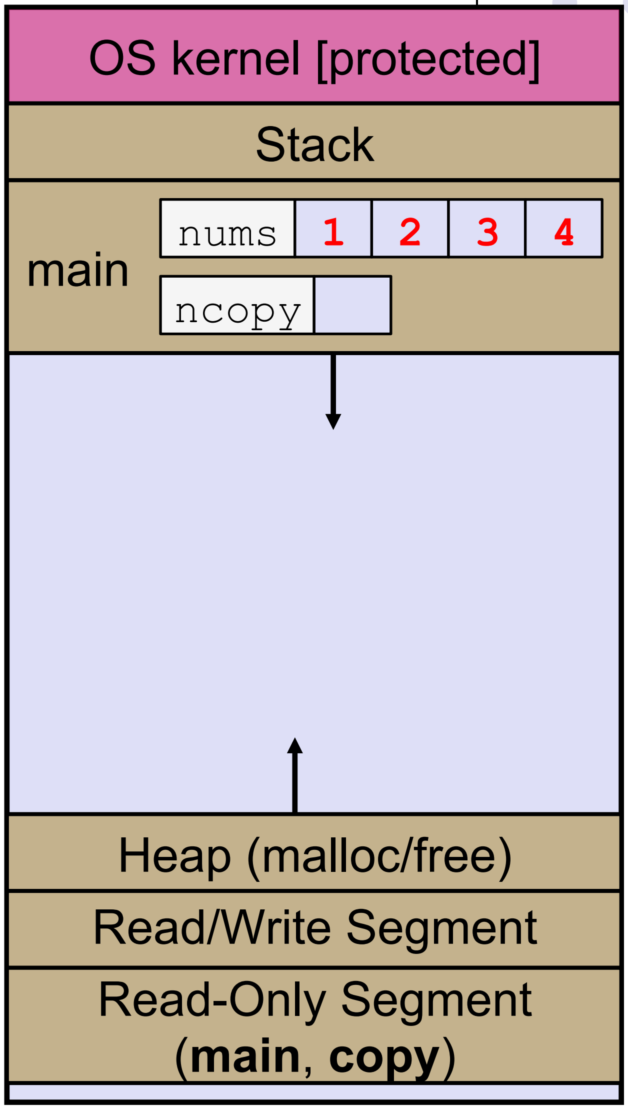
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (2/11)

arraycopy.c

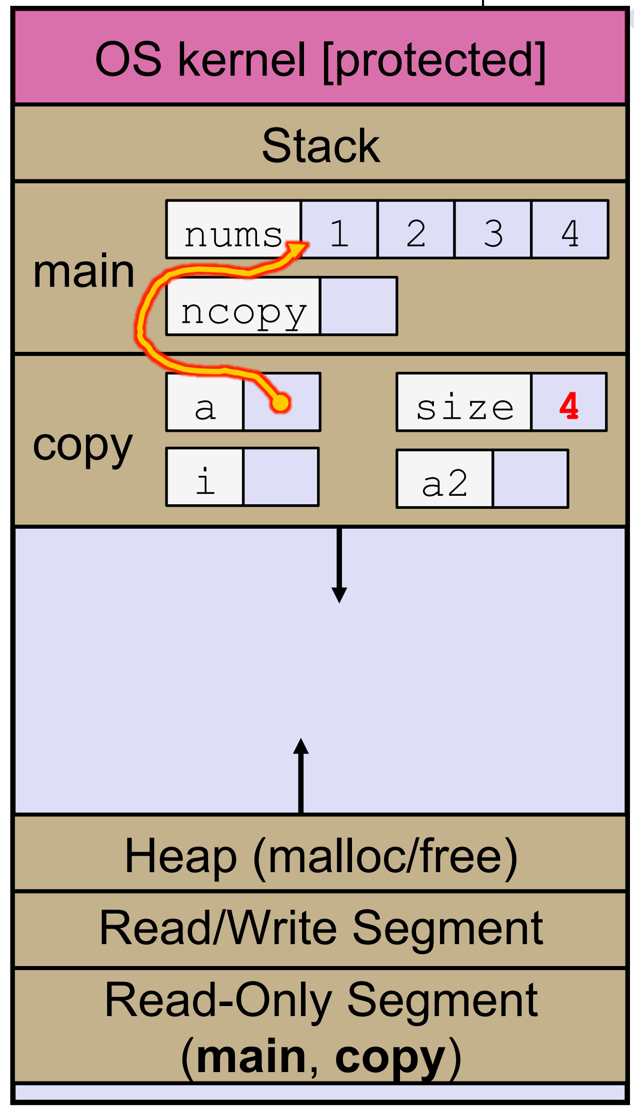
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (3/11)

arraycopy.c

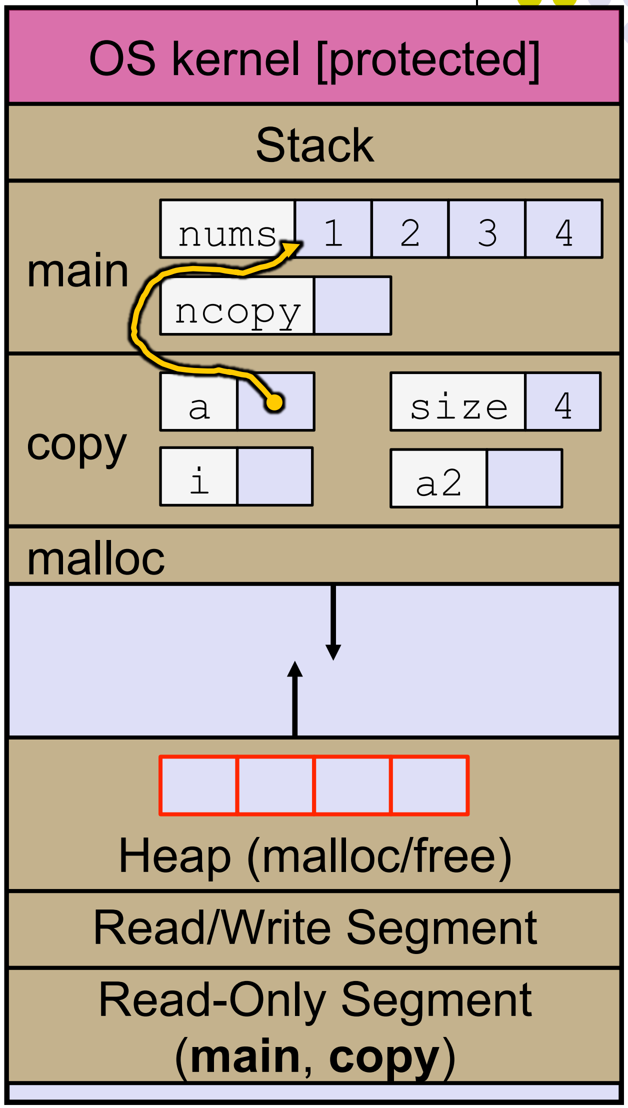
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (4/11)

arraycopy.c

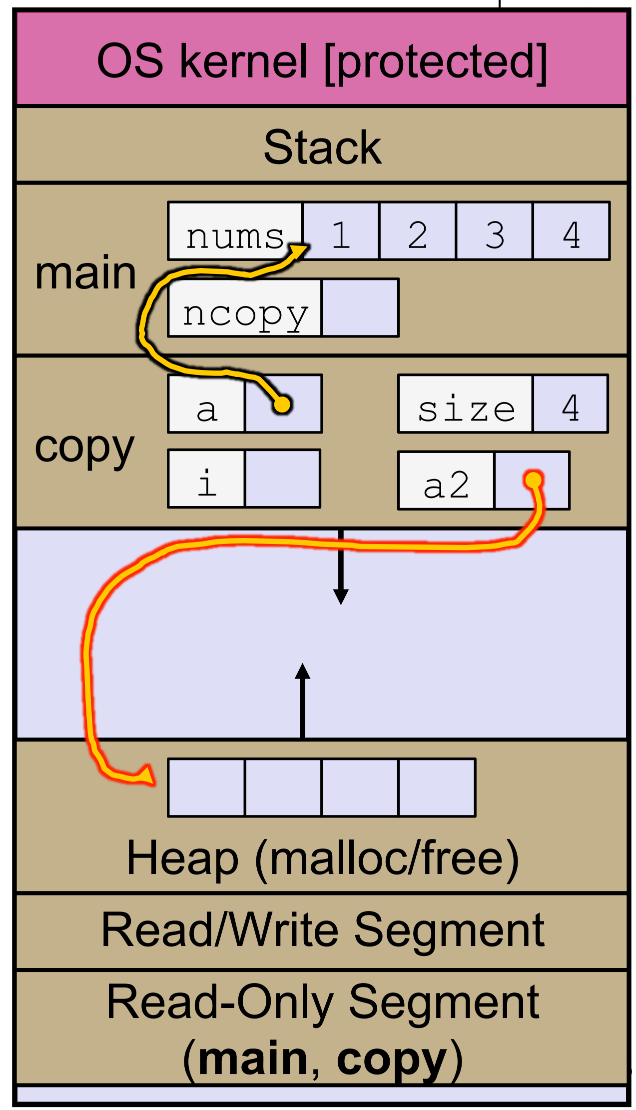
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (5/11)

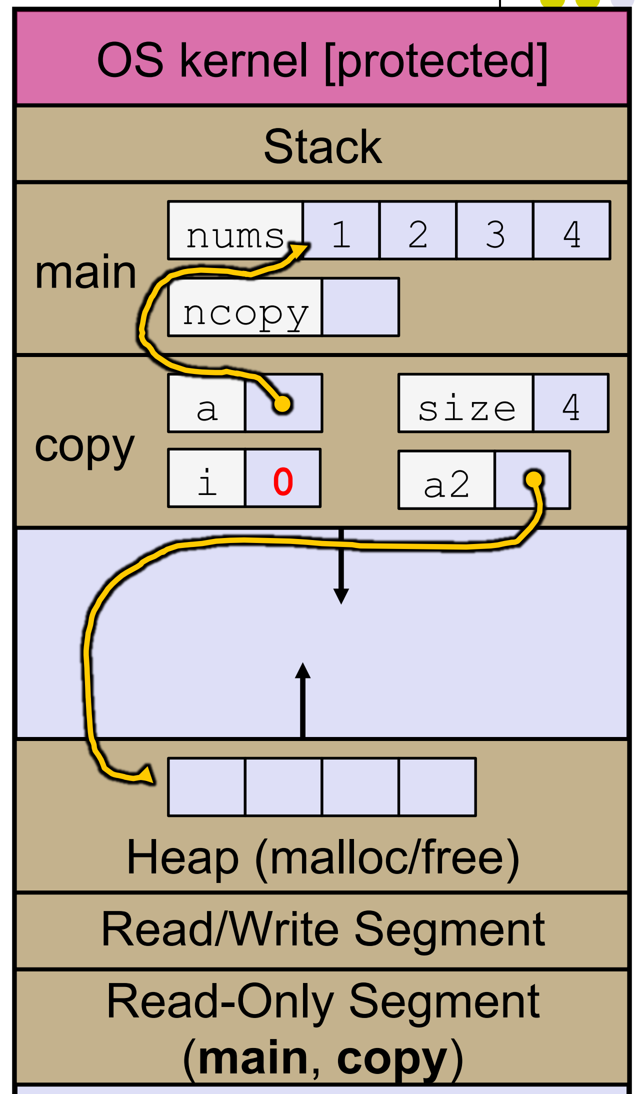
arraycopy.c

```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (6/11)

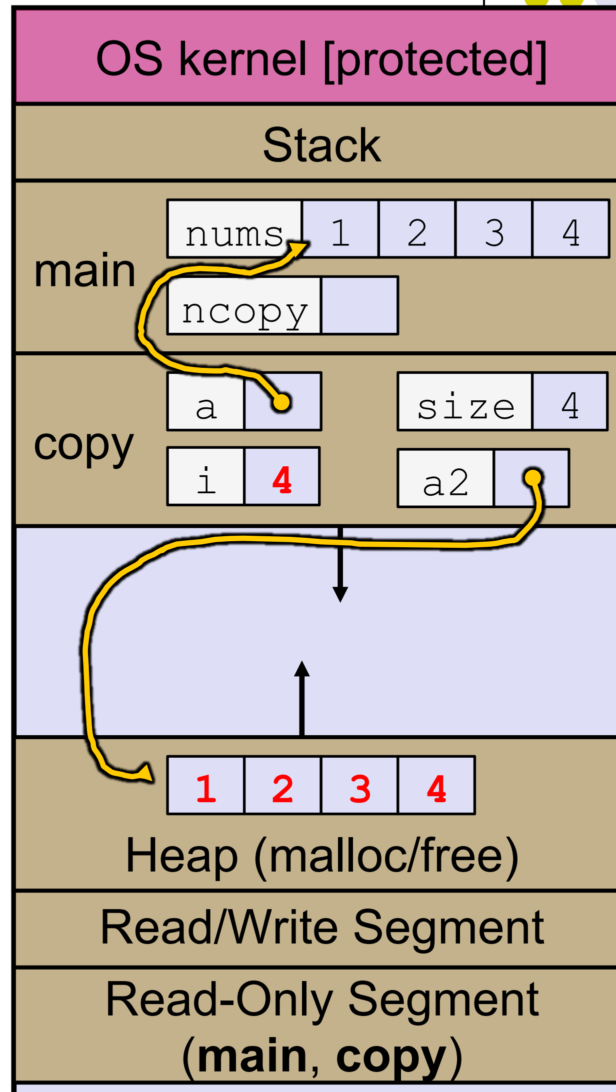
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (7/11)

arraycopy.c

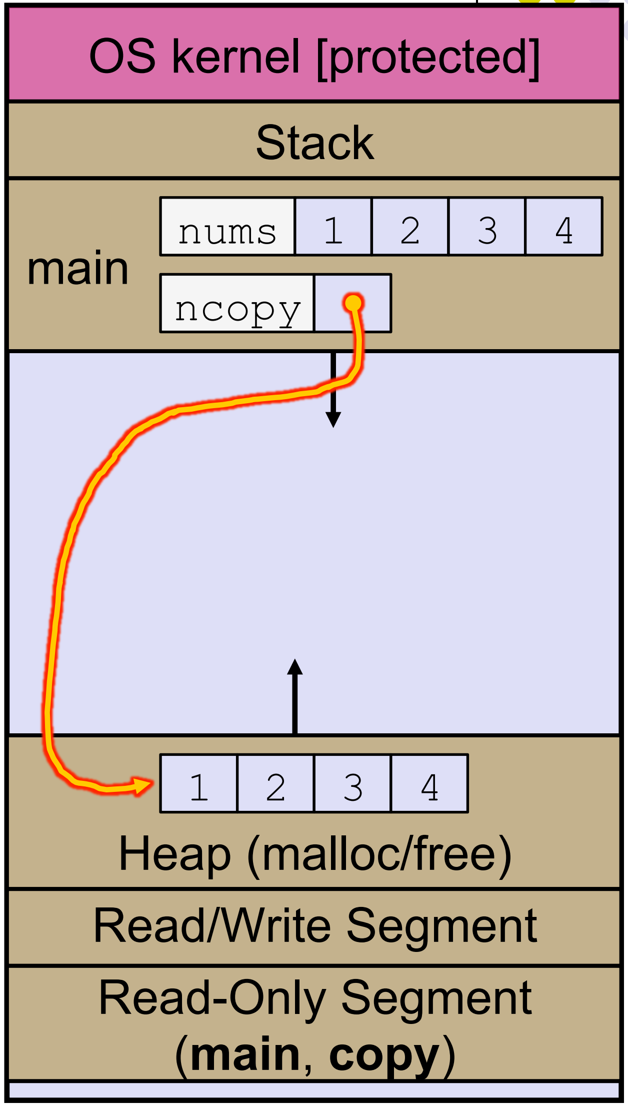
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (8/11)

arraycopy.c

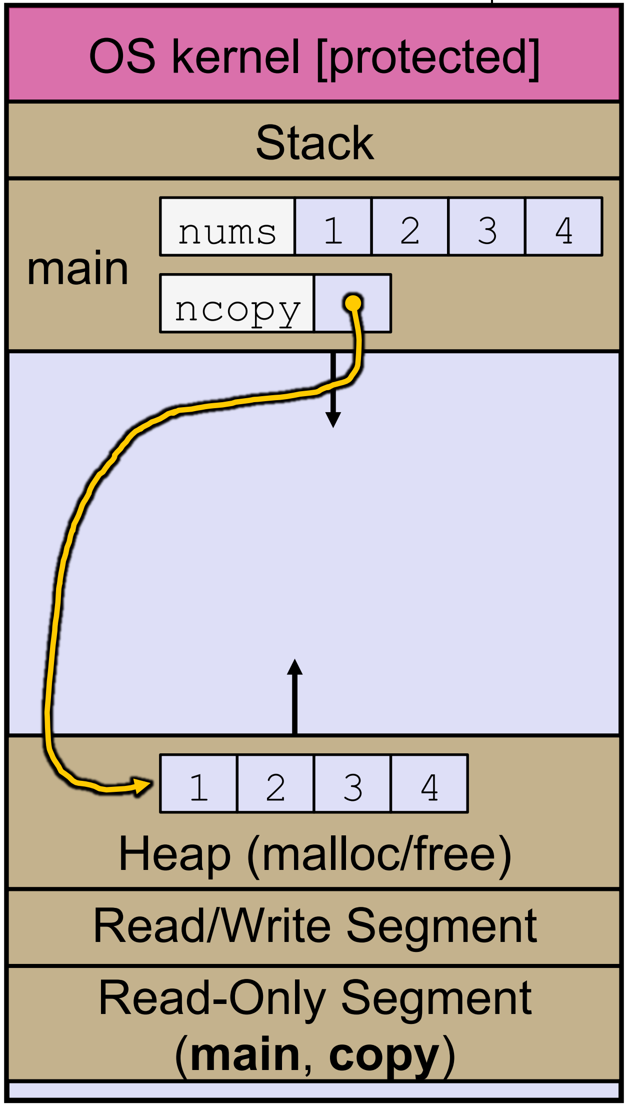
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (9/11)

arraycopy.c

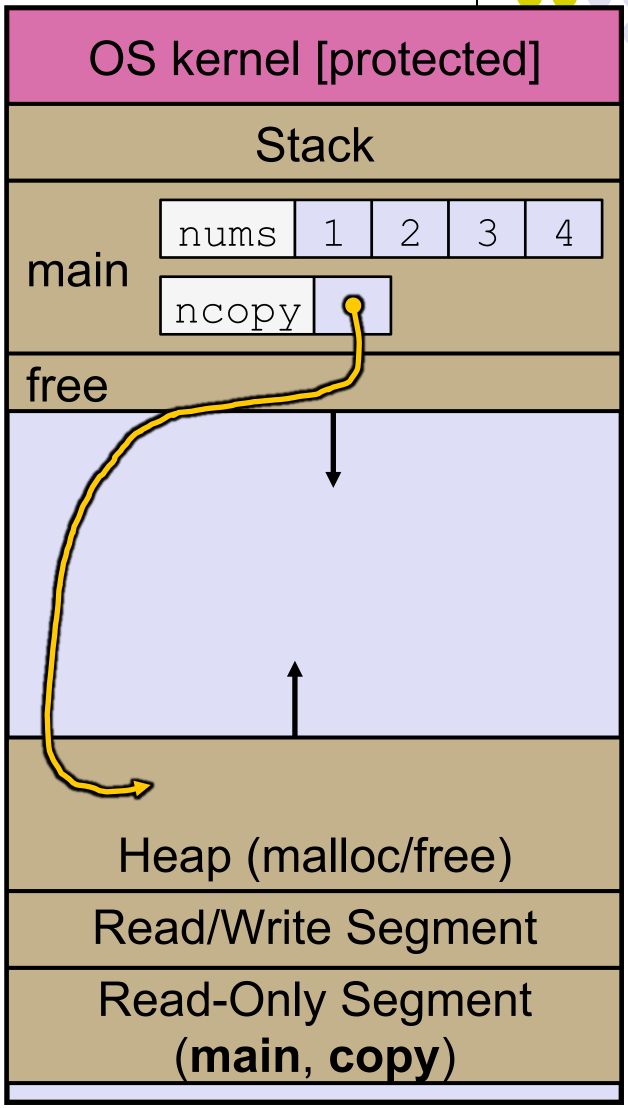
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (10/11)

arraycopy.c

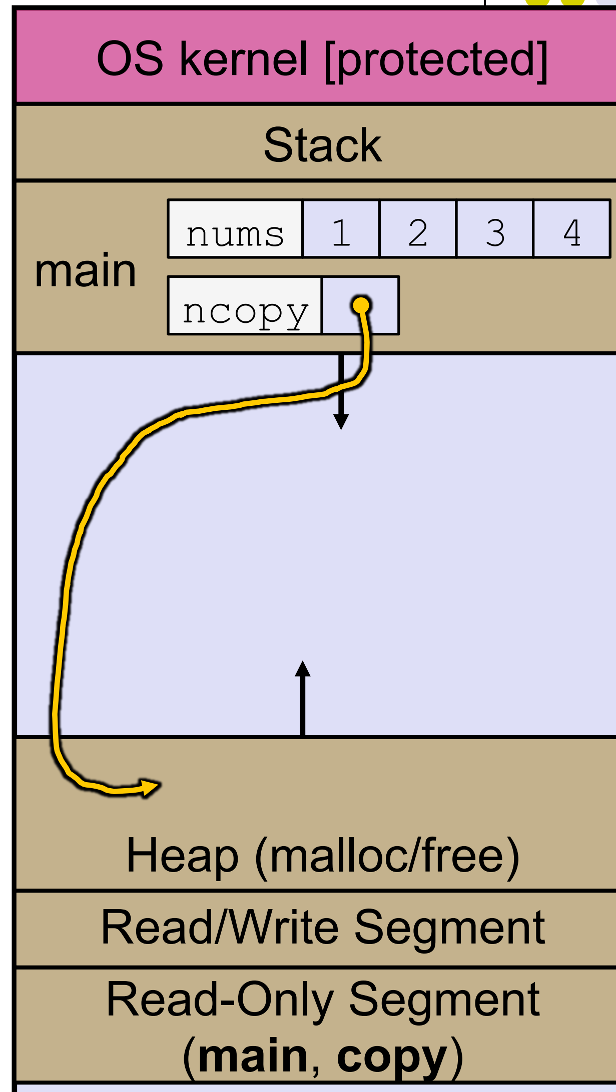
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (11/11)

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Memory Corruption

- There are all sorts of ways to corrupt memory in C

memcorrupt.c

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assign past the end of an array
  b[0] += 2;   // assume malloc zeros out memory
  c = b+3;     // mess up your pointer arithmetic
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed pointer

  // any many more!
  return 0;
}
```

Memory Leak (1/2)

- A memory leak occurs when
  - code fails to deallocate dynamically-allocated memory that is n o longer used
  - *e.g.* forget to `free` malloc-ed block, lose/change pointer to m alloc-ed block

Memory Leak (2/2)

- <u>Implication</u>: program's VM footprint will keep growing
  - This might be OK for *short-lived* program, since memory deallocated when program ends
  - Usually has bad repercussions for *long-lived* programs
    - Might slow down over time (*e.g.* lead to VM thrashing)
    - Might exhaust all available memory and crash
    - Other programs might get starved of memory

프로그램 11-1

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

int main()
{
    int *p;     // 동적 할당된 블록을 가리킬 포인터
    int n, i;

    printf("몇 개의 정수를 입력하고 싶으십니까? ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("오류: 정수 개수를 잘못 입력하였습니다. ");
        printf("프로그램을 종료합니다.\n");
        return -1;
    }

    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        printf("오류: 메모리가 부족합니다. ");
        printf("프로그램을 종료합니다.\n");
        return -1;
    }

    printf("%d 개의 정수를 입력해 주세요: ", n);
    for (i = 0; i < n; ++i)
        scanf("%d", &p[i]);
    printf("당신이 입력한 정수를 역순으로 출력합니다: ");
    for (i = n-1; i >= 0; --i)
        printf("%d ", p[i]);
    printf("\n");

    return 0;
}
```

입력한 n이 올바른 값인지 검사

입력할 정수를 저장할 변수를 동적으로 할당함

malloc이 성공적으로 메모리를 할당하였는지 검사

assert 매크로

- assert
  - 디버깅할 때에 특정 조건을 검사하기 위한 매크로
  - `assert(수식);` 형태로 사용함
  - 프로그램 수행 중 '수식'이 참이면 오류 메시지를 출력하고 프로그램을 종료시킴

- assert 활용 시 주의사항
  - `assert` 매크로는 전처리기를 통해 무시하도록 조정할 수 있음
  - NDEBUG(no debug)를 선언하면 `assert` 매크로를 무시함
    - IDE 환경에서 Release 모드로 프로젝트 설정
    - 명령줄 환경에서: `cl -DNDEBUG 파일이름`
  - 요즘엔 소프트웨어 안정성을 위해, 디버깅이 끝난 후에도 `assert`를 생략하지 않도록 권유하고 있음

프로그램 11-2

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

int main()
{
    int *p;     // 동적 할당된 블록을 가리킬 포인터
    int n, i;

    printf("몇 개의 정수를 입력하고 싶으십니까? ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("오류: 정수 개수를 잘못 입력하였습니다. ");
        printf("프로그램을                    );
        return -1;
    }

    p = (int *) malloc(n * sizeof(int));
    assert(p != NULL);

    printf("%d 개의 정수를 입력해 주세요: ", n);
    for (i = 0; i < n; ++i)
        scanf("%d", &p[i]);
    printf("당신이 입력한 정수를 역순으로 출력합니다: ");
    for (i = n-1; i >= 0; --i)
        printf("%d ", p[i]);
    printf("\n");

    return 0;
}
```

malloc 리턴값을 assert로 검사함
(오류 발생을 위한 표준 입력 값 예시: 1234567890123)

Questions?