CSCI-3753: Operating Systems Fall 2019

Abigail Fernandes

Department of Computer Science

University of Colorado Boulder

Week 11
> Programming Assignment 4
> Memory Management

Assignment Goal

Implement a paging strategy that a paging simulator can use to maximize the performance of the memory access in a set of pre-defined programs

Action items

- Implement **LRU** algorithm

  `pager-lru.c`
- Implement any form of **predictive paging algorithm**

  `pager-predict.c`

Paging Simulator

- Run a random set of 5 pre-defined programs utilizing a limited number of shared physical pages

- Provided default values in `simulator.h`
  - 20 virtual pages per process (MAX_PROC_PAGES)
  - 100 physical pages (frames) in total (PHYSICAL_PAGES)
  - 20 simultaneous processes competing for pages (MAX_PROCESSES)
  - 128 memory unit page size (PAGE_SIZE)
  - 100 tick delay to swap a page in or out (PAGE_WAIT)
    - Each instruction or step in the simulated programs requires 1 tick to complete.

Paging Simulator

- Is the environment resource constrained?
- How many physical pages can be swapped in at a given time?
- How many virtual pages will you have to access at most?
- Swapping a page in and out takes how much time?

Paging Simulator

Provide 3 functions for interaction

- To control the allocation of virtual and physical pages
  - `pagein()`
  - `pageout()`
- To handle the page fault
  - **`pageit()`** ← core paging function that needs implementation

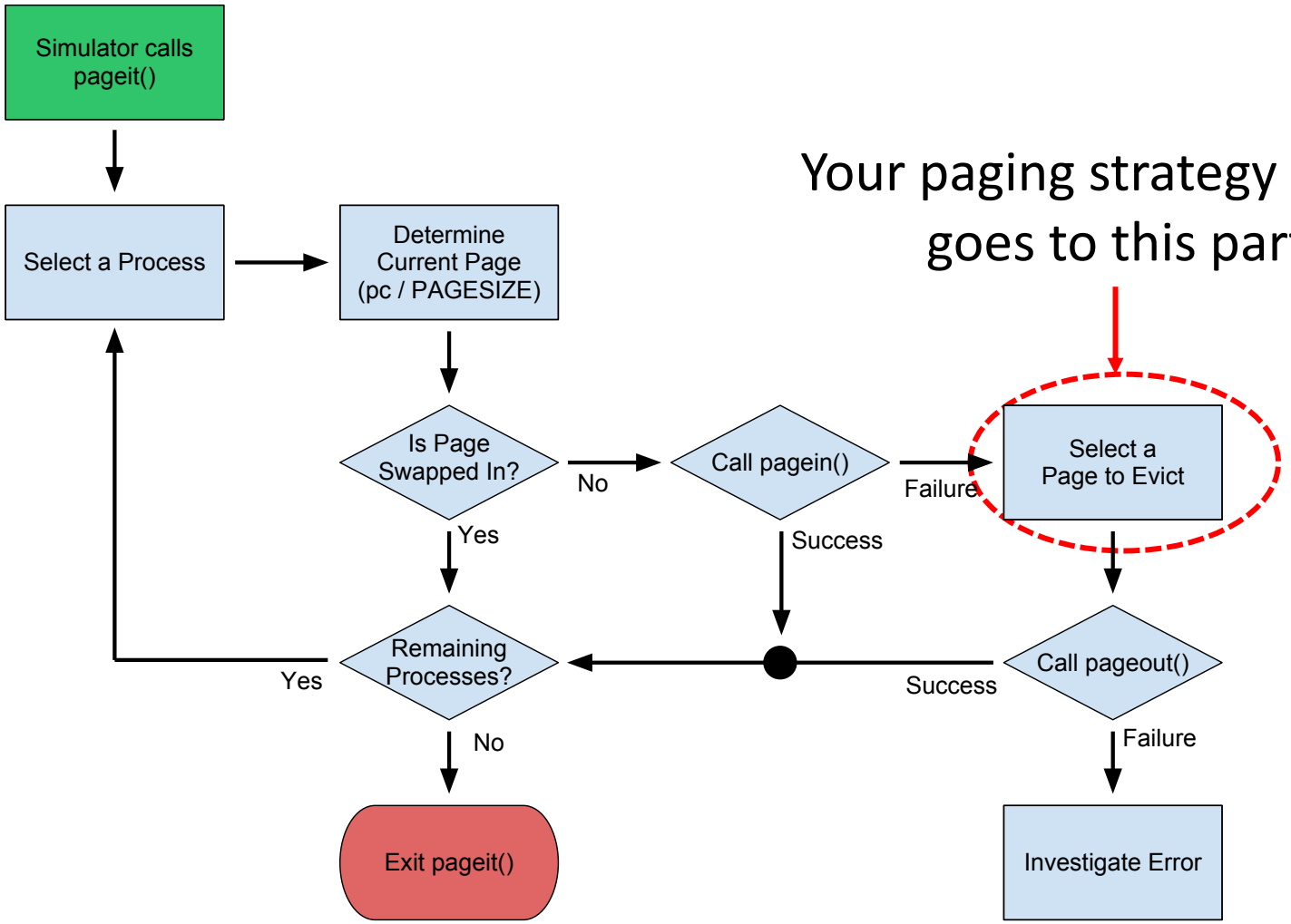
Source code is provided.

- `simulator.h, simulator.c`
- `pager-basic.c, pager-lru.c, pager-predict.c`

pager-basic.c

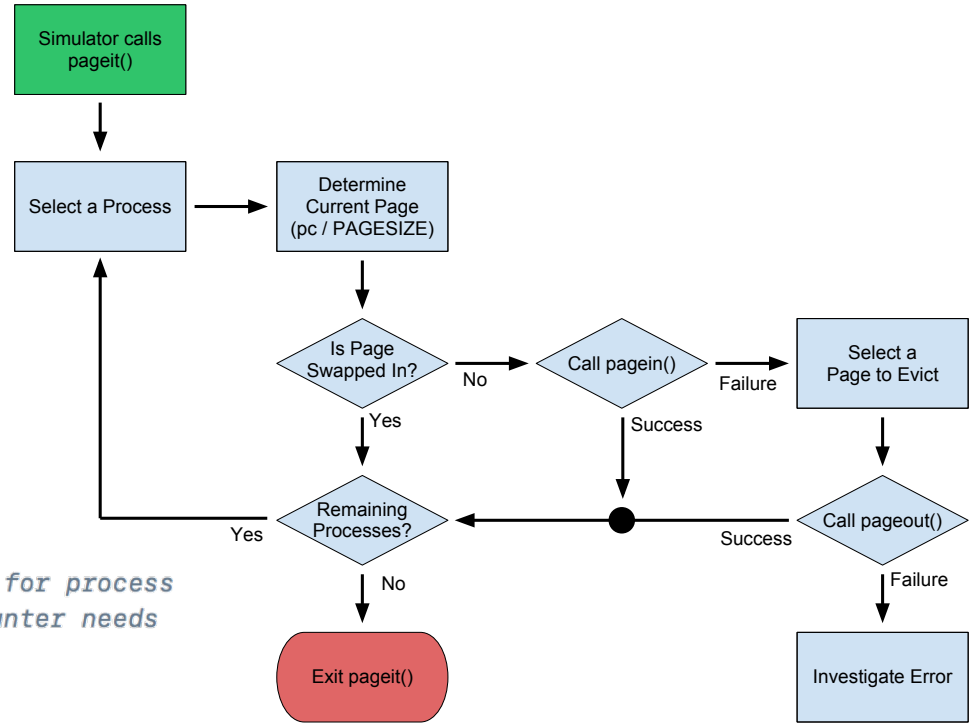
- A basic “one-process-at-a-time” implementation
- A simple demonstration of the simulator API
- Doesn’t need any implementation from YOU!!!

pager-basic.c



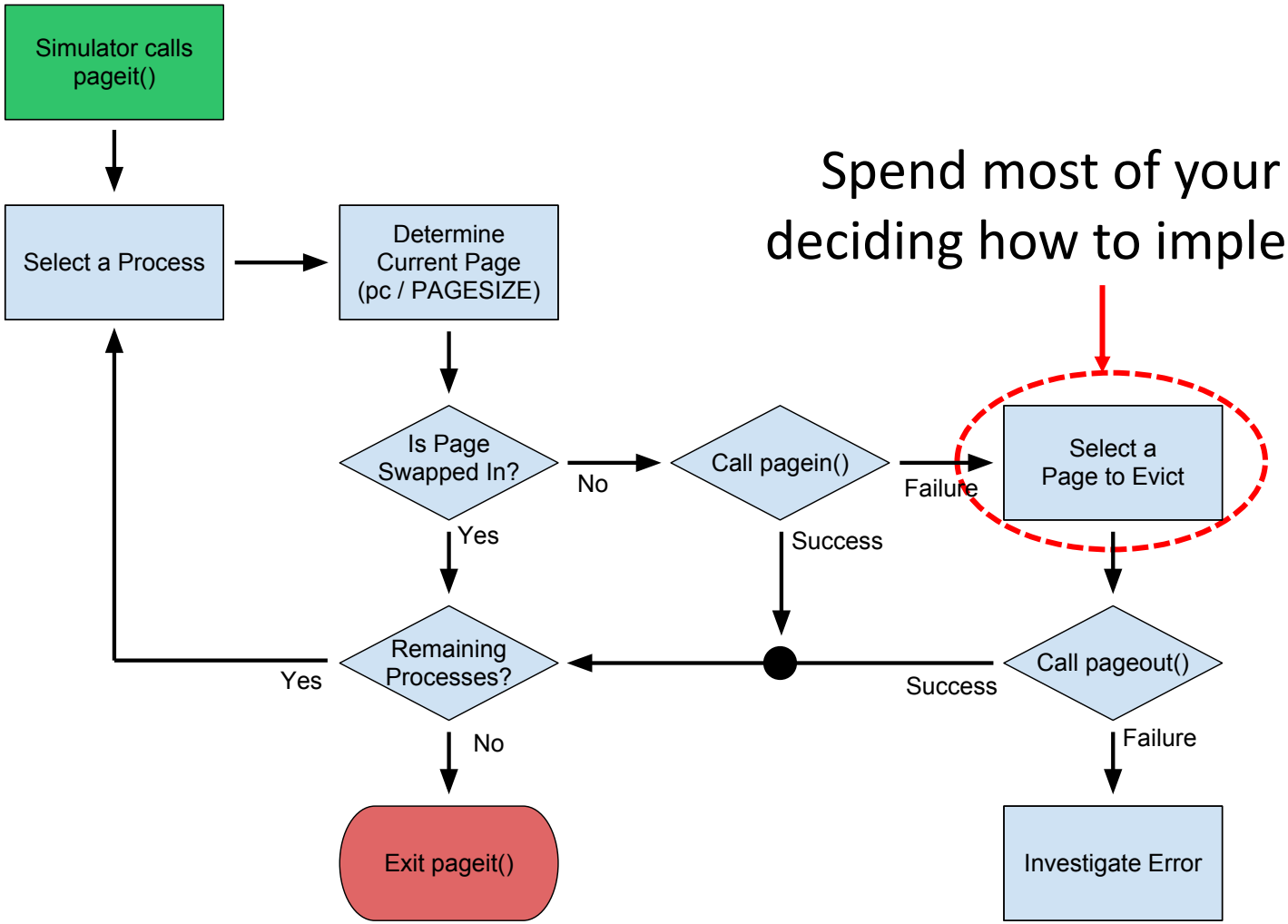
Your paging strategy mostly goes to this part.

```c
#include "simulator.h"

void pageit(Pentry q[MAXPROCESSES]) {

    /* Local vars */
    int proc;
    int pc;
    int page;
    int oldpage;

    /* Trivial paging strategy */
    /* Select first active process */
    for(proc=0; proc<MAXPROCESSES; proc++) {
    /* Is process active? */
    if(q[proc].active) {
        /* Dedicate all work to first active process*/
        pc = q[proc].pc;                // program counter for process
        page = pc/PAGESIZE;          // page the program counter needs
        /* Is page swaped-out? */
        if(!q[proc].pages[page]) {
        /* Try to swap in */
        if(!pagein(proc,page)) {
            /* If swapping fails, swap out another page */
            for(oldpage=0; oldpage < q[proc].npages; oldpage++) {
            /* Make sure page isn't one I want */
            if(oldpage != page) {
                /* Try to swap-out */
                if(pageout(proc,oldpage)) {
                /* Break loop once swap-out starts*/
                break;
                }
            }
            }
        }
        }
        /* Break loop after finding first active process */
        break;
    }
    }
}
```



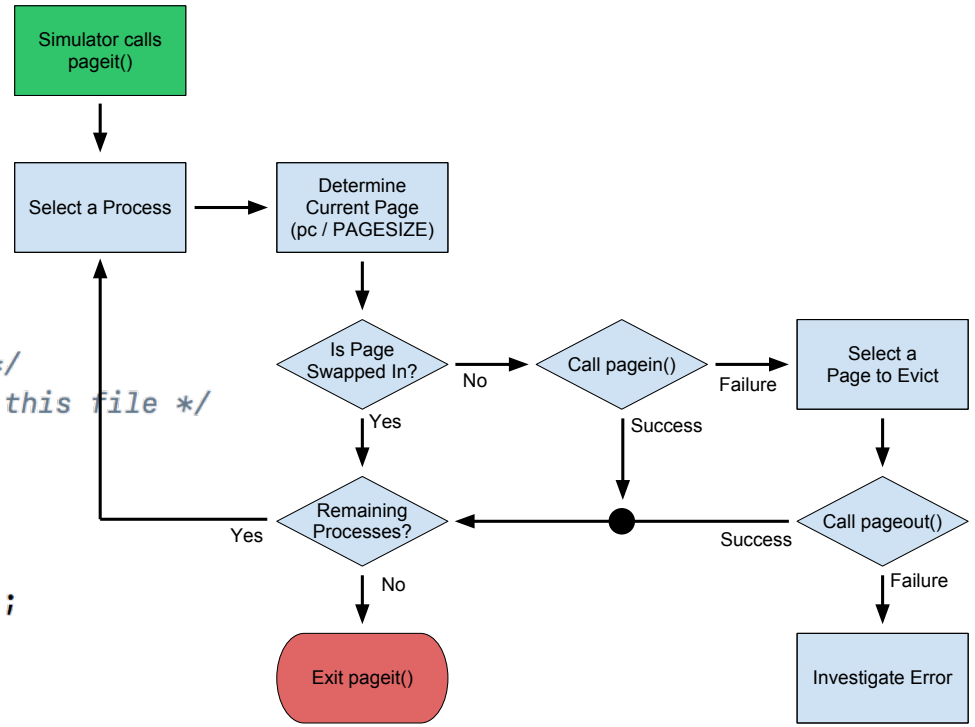
pager-basic.c

pager-lru.c



Spend most of your time deciding how to implement it

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

#include "simulator.h"

void pageit(Pentry q[MAXPROCESSES]) {

    /* This file contains the stub for an LRU pager */
    /* You may need to add/remove/modify any part of this file */

    /* Static vars */
    static int initialized = 0;
    static int tick = 1; // artificial time
    static int timestamps[MAXPROCESSES][MAXPROCPAGES];

    /* Local vars */
    int proctmp;
    int pagetmp;

    /* initialize static vars on first run */
    if(!initialized){
    for(proctmp=0; proctmp < MAXPROCESSES; proctmp++){
        for(pagetmp=0; pagetmp < MAXPROCPAGES; pagetmp++){
        timestamps[proctmp][pagetmp] = 0;
        }
    }
    initialized = 1;
    }

    /* TODO: Implement LRU Paging */
    fprintf(stderr, "pager-lru not yet implemented. Exiting...\n");
    exit(EXIT_FAILURE);

    /* advance time for next pageit iteration */
```



pager-lru.c

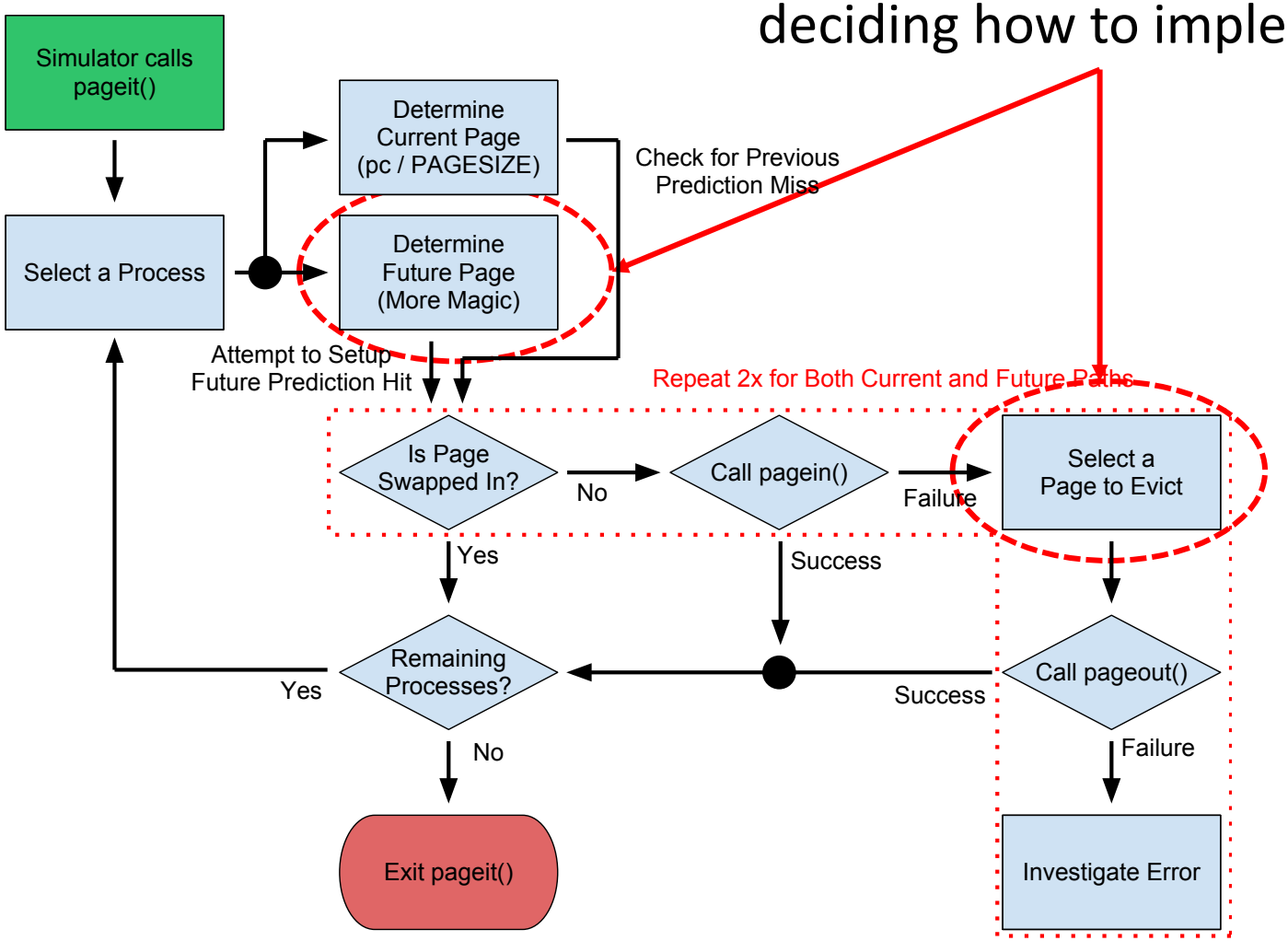
pager-predict.c

- Require a predictive algorithm that
  - Attempts to predict what pages each process will require in the future and then
  - Swaps these pages in before they are needed

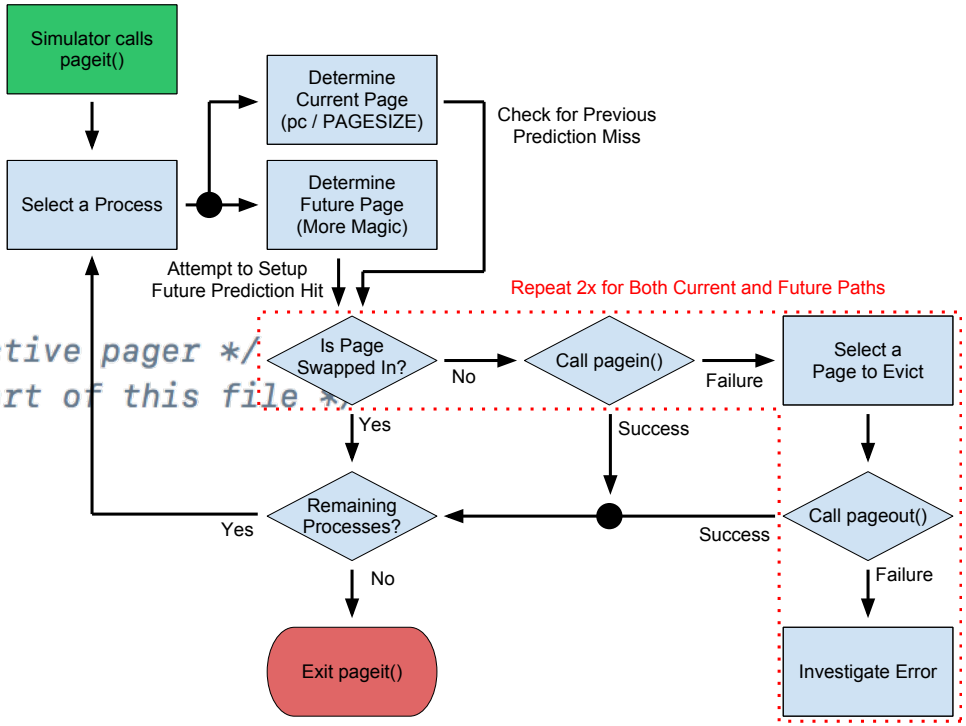
- **Note**: In any predictive operation, you ideally wish to stay 100-200 ticks ahead of the execution of each process.

pager-predict.c

Spend most of your time deciding how to implement it



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

#include "simulator.h"

void pageit(Pentry q[MAXPROCESSES]) {

    /* This file contains the stub for a predictive pager */
    /* You may need to add/remove/modify any part of this file */

    /* Static vars */
    static int initialized = 0;
    static int tick = 1; // artificial time

    /* Local vars */


    /* initialize static vars on first run */
    if(!initialized){
    /* Init complex static vars here */

    initialized = 1;
    }

    /* TODO: Implement Predictive Paging */
    fprintf(stderr, "pager-predict not yet implemented. Exiting...\n");
    exit(EXIT_FAILURE);

    /* advance time for next pageit iteration */
    tick++;
}
```

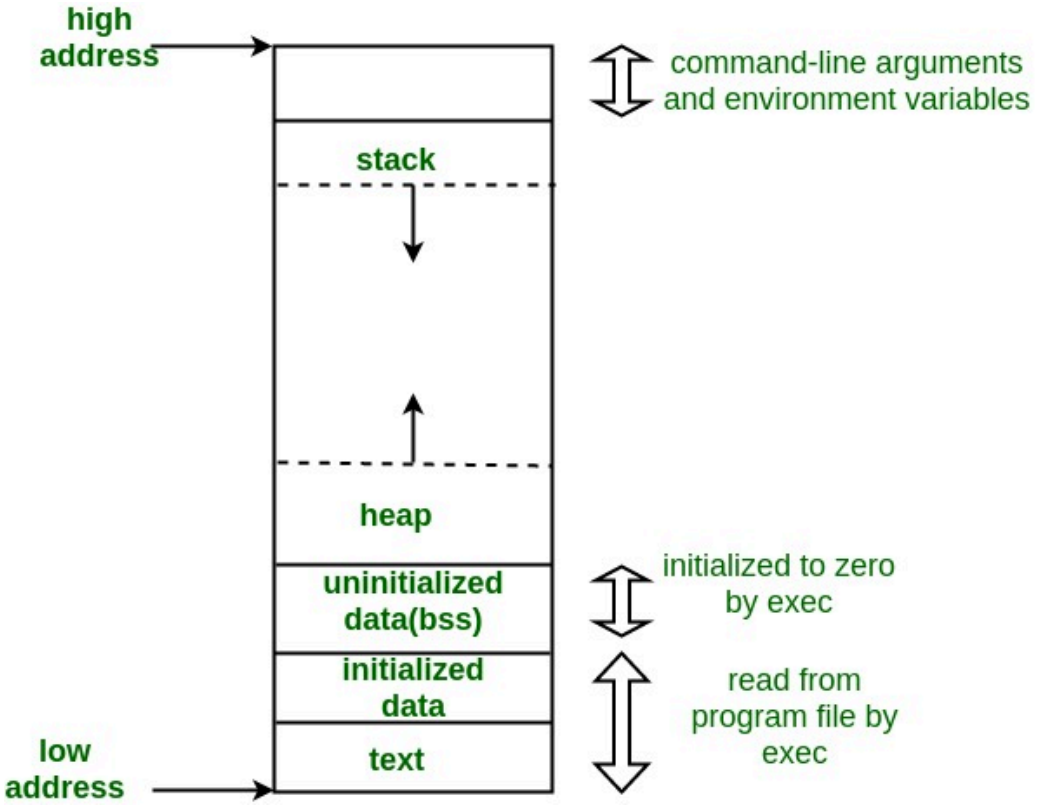


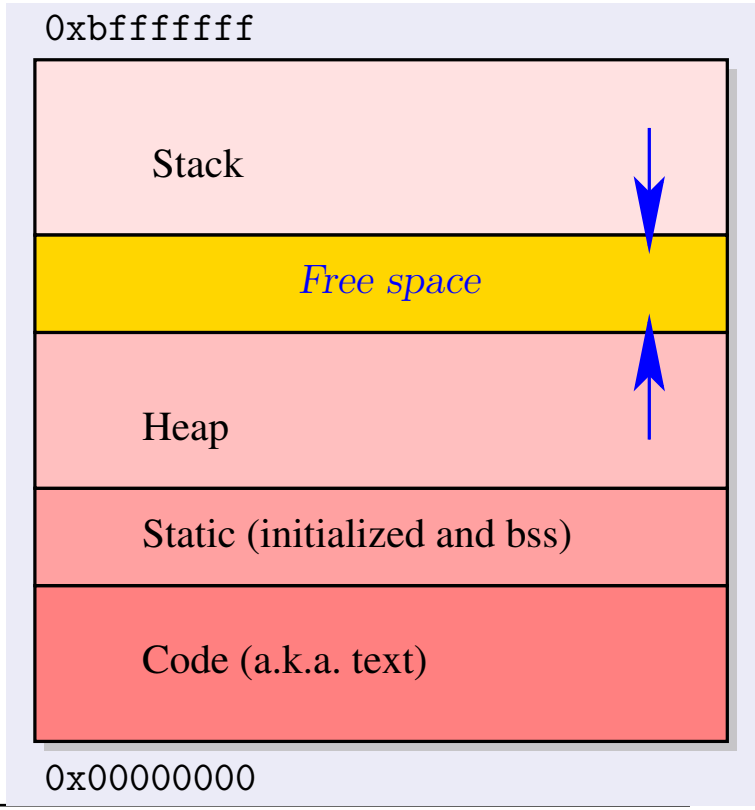
pager-predict.c

Week 12
> Programming Assignment 4
> Memory Management

Memory Address Space of a Process

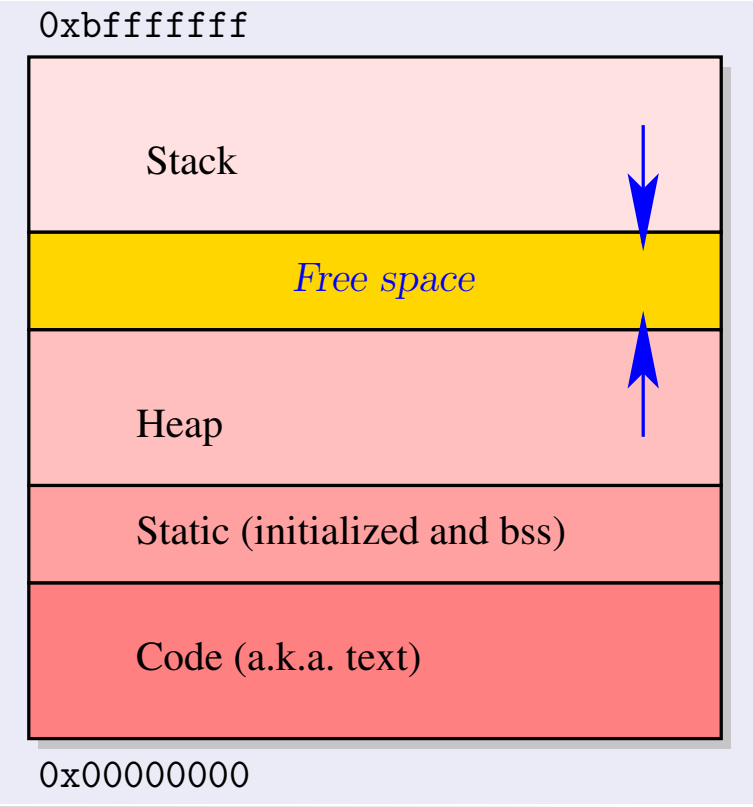
- Allocated and initialized when loading and executing





Memory Address Space of a Process

- Allocated and initialized when loading and executing a program
- Consists of
  - Code segment
    - .o and executable code
    - Below the heap or stack in order to prevent heaps and stack overflows from overwriting it
    - Usually
      - Sharable
      - Read-only
      - Fixed size

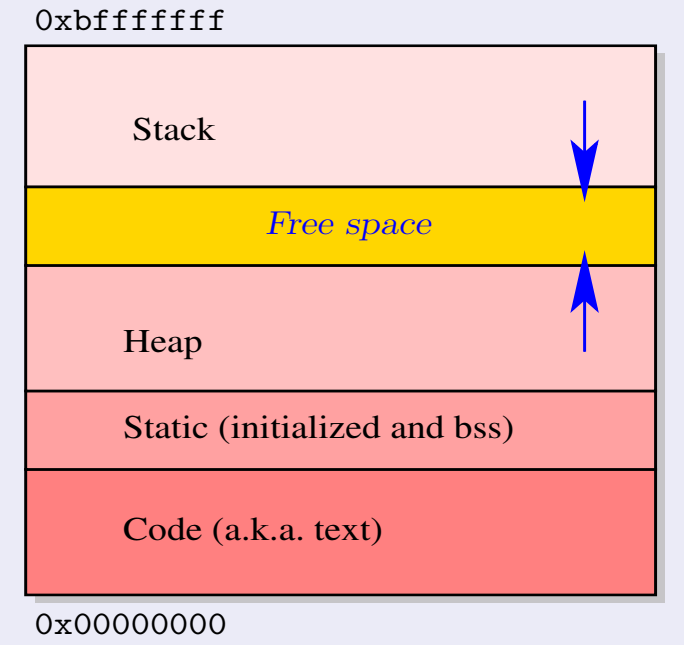


Memory Address Space of a Process

- Allocated and initialized when loading and executing a program
- Consists of
  - Static segments
    - **Data segment**

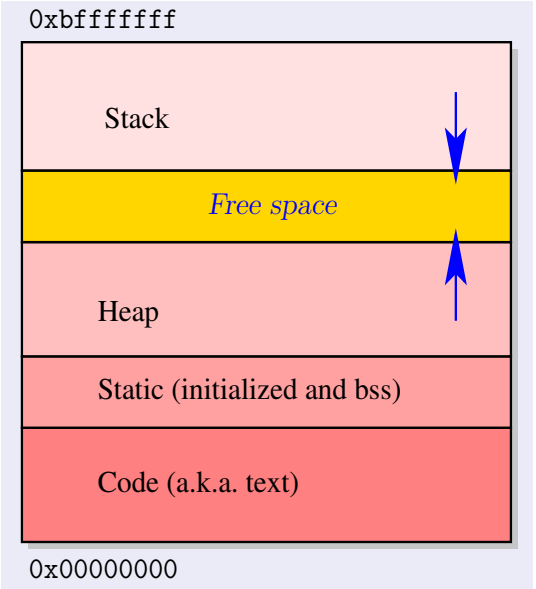
      Initialized global and C static Variables
    - **BSS segment**

      Uninitialized global variables that are zeroed when initializing the process, are stored in .bss segment

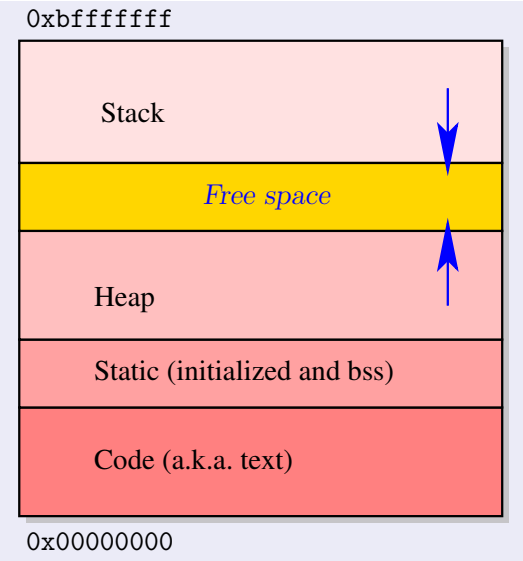


Memory Address Space of a Process

- Allocated and initialized when loading and executing a program
- Consist of
  - Stack segment
    - Stack frames of function call arguments and local variables, also called automatic variables in C
  - Heap segment
    - Dynamic allocation
      - malloc, realloc, and free,
      - brk and sbrk system calls to adjust its size



System Calls

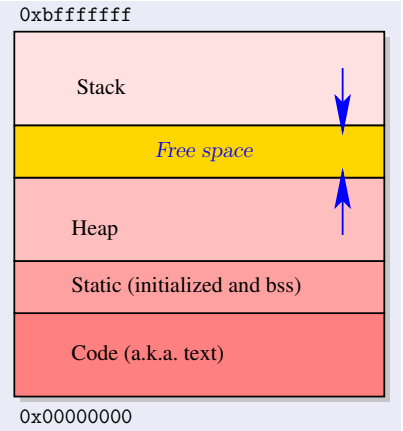


- To change data segment file
  - `brk()`
  - `sbrk()`

- Program break is the first location after the end of the uninitialized bss segment.
  - Increasing the program break has the effect of allocating memory to the process
  - Decreasing the break deallocates memory.

- Both functions are deprecated as "programmer interface" functions.

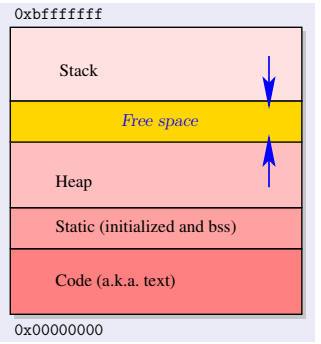
System Calls




```
int brk(void *addr);
```

Set the end of the data segment to the value specified by `addr`, when that value is reasonable, the system has enough memory and the process does not exceed its maximum data size

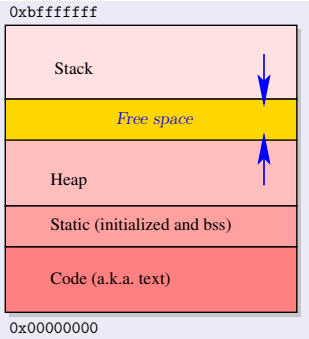
System Calls




```
void *sbrk(intptr_t increment);
```

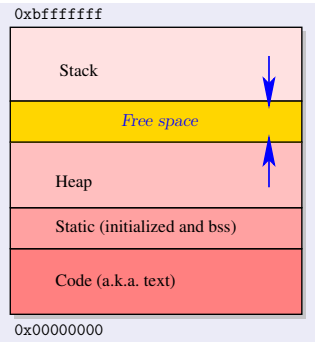
- sbrk() increments the program's data space by `increment` bytes.
- `sbrk(0)` returns the current location of the program break.

System Calls



- Both `brk()` and `sbrk()` extend heap ??
  - `brk(b)` sets the end of the heap to b
  - `sbrk(n)` extends the end of the heap by n bytes
  - `sbrk(0)` returns the virtual address just past the end of the heap.

System Calls



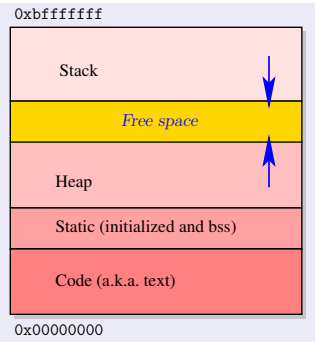

```
void* malloc(size_t size);
```

- Allocate `size` bytes of uninitialized storage
  - Call `sbrk()` to get the memory to allocate in the heap
- More efficient than allocating memory using simply `brk()` or `sbrk()`
  - Do buffering
  - However, does not always invoke `sbrk()`
  - When it calls `sbrk()`, it calls it to allocate a much larger memory than needed

Virtual Memory

- `getpagesize()`
- `getrlimit()` and `setrlimit()`
  - Process virtual memory size limit
  - Max CPU time
  - Max data segment size
  - …
- `pmap pid/pids`
  - report memory map of a process or processes
- `getconf`
  - get configuration values of standards set in the current operating environment

System Calls




```
void * mmap(void *start, size_t length, int
  prot, int flags, int fd, off_t offset)
```

Create a new mapping in the virtual address space of the calling process

Week 11 – Checklist

- ❑ Discuss PA4
- ❑ Discuss memory management