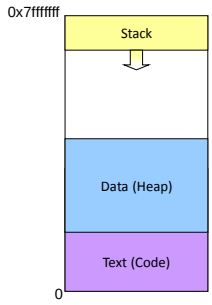
Final Review

Process's Address Space

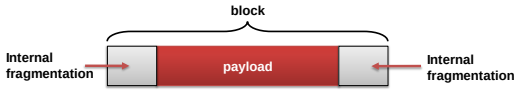


Goals of good malloc/free

- Primary
  - Good time performance for malloc and free
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - Good space utilization
    - User allocated structures should be large fraction of the heap
    - Want to minimize "fragmentation"
- Some Others
  - Good locality properties
    - Structures allocated close in time should be close in space
    - "Similar" objects should be allocated close in space
  - Robust
    - Can check that free(p1) is on a valid allocated object p1
    - Can check that memory references are to allocated space

Internal fragmentation

- Fragmentation causes poor memory utilization
  - Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - The difference between the block size and the payload size

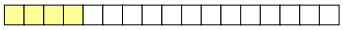


  - Due to:
    - overhead of maintaining heap data structures
    - padding for alignment purposes
    - explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of previous requests, so easy to measure

External fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough

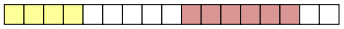
```
p1 = malloc(4)
```



```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```



```
p4 = malloc(6)
```

oops!

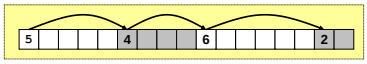
Depends on the pattern of *future* requests, so it's difficult to measure

What to blocks to allocate?

- **First fit:**
  - Search list from beginning, select first free block that fits
  - Can take linear time in num. of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- **Next fit:**
  - Like first-fit, but start from end of previous search
  - Research suggests that fragmentation is worse
- **Best fit:**
  - Search the list, select free block with closest size that fits
  - Keeps fragments small --- usually helps fragmentation
  - Will typically run slower than first-fit

Keeping track of free blocks

- **Implicit list** using lengths -- links all blocks



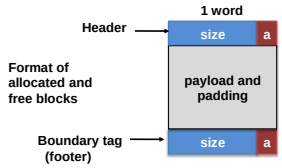
- **Explicit list** tracks only free blocks using pointers inside the free blocks

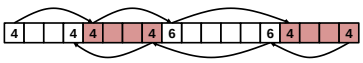
- **Segregated free list** – different free lists for different size classes
- **Blocks sorted by size**
  - Can use a balanced tree with pointers within each free block, and the length used as a key

Implicit list: Bidirectional coalescing

- **Boundary tags [Knuth73]**
  - Replicate size/allocated word at bottom of free blocks
  - Allows traversing a "list" backwards, but requires extra space
  - Important and general technique!

Donald Knuth 1938-



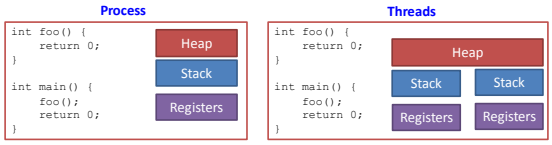


Concurrency and Parallelism

- Many programs need to perform mostly independent tasks that do not need to be serialized
  - Web server: page requests
  - Text editor: auto-save, spell checking, text entry
  - Web client: tabbed browsing
- Concurrency
  - Multiple, generally different, tasks
  - For convenience
- Parallelism
  - Multiple copies of the same task
  - For performance
- This is one of the key problems we are facing today

Threads and Processes

- A process is different than a thread
  - Conceptually (On Linux it is more complicated)
- Thread: Separate execution streams in same address space
  - "Lightweight process"



- Can have multiple threads within a process

Threads and processes

- Most modern OS's support both
  - Process – defines address space and general process attributes
  - Thread – defines a sequential execution stream within the process context
- Threads are bound to 1 process/address space
  - Isolated from other threads in other processes
- Communicating between processes is hard
  - But communicating between threads is easy
    - Threads share access to same global address space
    - Can read/write same global variables

Complications with threads

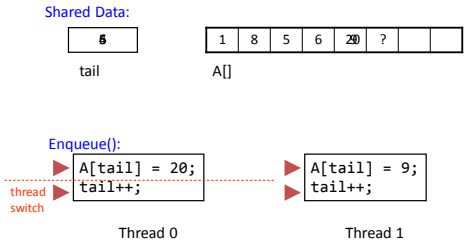
- Different from fork()/exec()
  - Fork creates a duplicate copy
  - State is replicated, NOT shared
- Threads share all global state
  - Global variables, static variables
  - Open files, open network connections
  - Signals
- What to do about the stack?

POSIX Threads

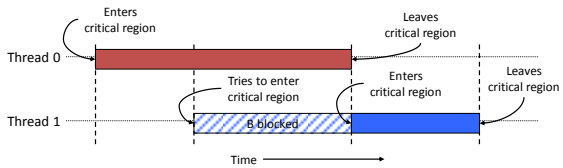
- Common on UNIX variants (Linux, Mac OS X)

```
int pthread_create(pthread_t * thread,
    const pthred_attr_t * attr,
    void * (*start_routine)(void *),
    void * arg);

void pthread_exit(void * value_ptr);

int pthread_join(pthread_t thread, void ** value_ptr);

int pthread_yield(void);
```

Race Condition



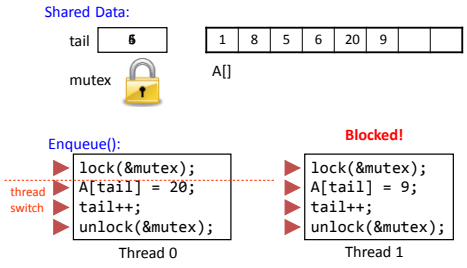
Critical Regions



Mutex

- MUTual EXclusion
- A mutex is a lock that only one thread can acquire
- All other threads attempting to enter the critical region will be blocked

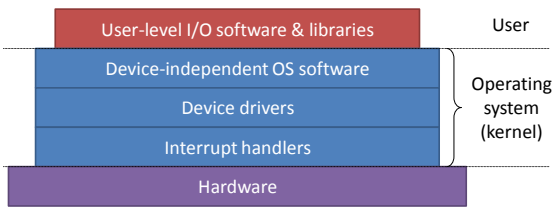
Critical Sections



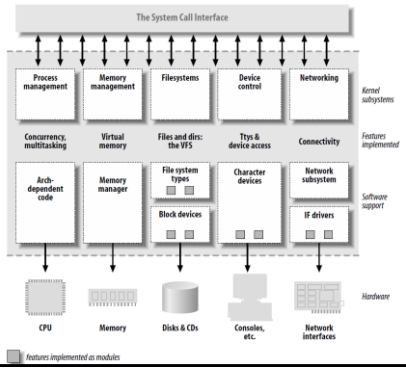
Deadlocks

- "A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause."
- Caused when:
  1. Mutual exclusion
  2. Hold and wait
  3. No preemption of resource
  4. *Circular wait*

Software Layers



Abstraction via the OS



Types of Devices

- **Block Devices**
  - A device that stores data in fixed-sized blocks, each uniquely addressed, and can be randomly accessed
  - E.g., Disks, Flash Drives
- **Character Devices**
  - Device that delivers or accepts a stream of characters
  - E.g., Keyboard, mouse, terminal

Mechanism vs. Policy

- Mechanism – What capabilities to have
- Policy – How to use a mechanism

- Drivers should be flexible by only providing mechanisms not policies
- Policies should generally be pushed up as much as possible

UNIX file operations

- File operations are implemented inside the kernel
  - Kernel maps operations to the specific file in question
  - Data flows from process memory to/from files on disk
  - But now we have a better picture…

- Data path: process ←→kernel ←→ disk
  - Kernel can intercept data before it reaches the disk
  - Easy mechanism for copying to/from kernel space (device drivers)

/dev

- Character and block devices can be exposed via a filesystem
- `/dev/` typically contains "files" that represent the different devices on a system

- `/dev/console` – the console
- `/dev/fd/` – a process's open file descriptors
- /dev/hda – first hard disk

Method of operation

- Device drivers respond to requests
  - They are part of the kernel
  - Interrupts from hardware devices
  - Requests from user space
- During module initialization you register for certain events
  - Interrupts from a specific device
  - File operations on a specific file system
- The Kernel's main responsibility is to route events to the correct device drivers

Module linking

- The module calls the kernel level API
- The kernel calls the module init function
- There must be a linking step
  - Kernel must be able to find init()
  - Module must be able to find kernel API functions
- When a module is loaded the kernel performs a linking operation on the module
  - Similar to user level linking (ELF format)