Operating Systems: Short notes

(Abhyuday Pandey BT/CSE/IITK)

Process

- The process is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its address space, the contents of the CPU registers (including the program counter and stack pointer, among others), and information about I/O (such as open files which can be read or written)

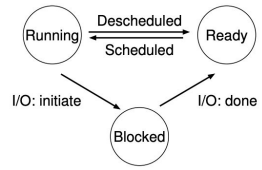


Figure 4.2: **Process: State Transitions**

- A process list contains information about all processes in the system. Each entry is found in what is sometimes called a process control block (PCB), which is really just a structure that contains information about a specific process.

Process API

`pid_t fork(void);`

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

> **fork**() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.
>
> The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.
>
> The child process is an exact duplicate of the parent process except for the following points (some ommitted):
>
> * The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.
>
> * The child's parent process ID is the same as the parent's process ID.
>
> * The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).
>
> * Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.
>
> * The child's set of pending signals is initially empty

(sigpending(2)).

When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point.

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

The **wait**() system call suspends execution of the calling process until one of its children terminates. The call wait(&status) is equivalent to:
waitpid(-1, &status, 0);
The **waitpid**() system call suspends execution of the calling process until a child specified by pid argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the options argument, as described below.
The value of pid can be:
< -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid. -1 meaning wait for any child process. 0 meaning wait for any child process whose process group ID is equal to that of the calling process. > 0 meaning wait for the child whose process ID is equal to the value of pid.

- exec()
    - it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the argv of that process.
    - Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

The Abstraction: Address Spaces

- when we describe the address space, what we are describing is the abstraction that the OS is providing to the running program.
- One major goal of a virtual memory (VM) system is **transparency** . The OS should implement virtual memory in a way that is invisible to the running program.
- Another goal of VM is efficiency. The OS should strive to make the virtualization as efficient as possible.
- Finally, a third VM goal is protection. The OS should make sure to protect processes from one another as well as the OS itself from processes.

Memory API

- In running a C program, there are **two types of memory** that are allocated
    - The first is called **stack memory (aka automatic memory)**, and allocations and deallocations of it are managed implicitly by the compiler for you.

    - **heap memory**, where all allocations and deallocations are explicitly handled by you.
- malloc()
    - malloc() returns a pointer to type void. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a cast
- free()
    - the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself
- brk
    - is used to change the location of the program's break: the location of the end of the heap.
    - It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break
    - An additional call sbrk is passed an increment but otherwise serves a similar purpose

---

```
int brk(void *addr);
void *sbrk(intptr_t increment);
On success, brk() returns zero. On error, -1 is returned.
On success, sbrk() returns the previous program break.(If the break was
increased, then this value is a pointer to the start of the newly allocated
memory).  On error, (void *) -1 is returned.
       brk() and sbrk() change the location of the program break, which
       defines the end of the process's data segment (i.e., the program
       break is the first location after the end of the uninitialized data
       segment).  Increasing the program break has the effect of allocating
       memory to the process; decreasing the break deallocates memory.

       brk() sets 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 (see
       setrlimit(2)).

       sbrk() increments the program's data space by increment bytes.
       Calling sbrk() with an increment of 0 can be used to find the current
       location of the program break.
```

- mmap
    - can create an **anonymous** memory region within your program — a region which is not associated with any particular file but rather with **swap space**. This memory can then also be treated like a heap and managed as such.

```
void *mmap(void *addr, size_t length, int prot, int flags,
                    int fd, off_t offset);
      On success, mmap() returns a pointer to the mapped area. On error,
             the value MAP_FAILED (that is, (void *) -1) is returned
```

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the kernel will pick a nearby page boundary (but always above or equal to the value specified by */proc/sys/vm/mmap_min_addr*) and attempt to create the mapping there. If another mapping already exists there, the kernel picks a new address that may or may not depend on the hint. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

After the mmap() call has returned, the file descriptor, *fd*, can be closed immediately without invalidating the mapping.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file).

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.

Memory mapped by mmap() is preserved across fork(2), with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

Mechanism: Limited Direct Execution

- By time sharing the CPU virtualization is achieved.
- The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

-
- **Restricted Operations**
    - A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.
    - Thus, the approach we take is to introduce a new processor mode, known as user mode; code that runs in user mode is restricted in what it can do
    - In contrast to user mode is kernel mode, which the operating system (or kernel) runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.
    - system calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.
    - To execute a system call, a program must execute a special **trap** instruction.
    - this instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode
    - When finished, the OS calls a special **return-from-trap** instruction
    - the processor will push the program counter, flags, and a few other registers onto a per-process kernel stack; the **return-from-trap** will pop these values off the stack and resume execution of the usermode program
    - how does the trap know which code to run inside the OS?
        - The kernel does so by setting up a trap table at boot time.

- One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

- Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted
- To specify the exact system call, a system-call number is usually assigned to each system call.
- This level of indirection serves as a form of protection; user code cannot specify an exact address to jump to, but rather must request a particular service via number.
-

- Switching Between Processes
  - How can the operating system regain control of the CPU so that it can switch between processes?
    - Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making system calls.
    - Systems like this often include an explicit yield system call, which does nothing except to transfer control to the OS so it can run other processes.

        - Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a **trap** to the OS.
    - A Non-Cooperative Approach: The OS Takes Control
        - timer interrupt
        - A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs.
        - OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that.
        - also during the boot sequence, the OS must start the timer, privileged op.
        - hardware has to save **enough** of the state of the program that was running when the interrupt occurred
    - Saving and Restoring Context
        - scheduler
        - If the decision is made to switch, the OS then executes a low-level piece of code which we refer to as a context switch.
            - OS has to do is save a few register values for the currently-executing process (onto its kernel stack, for example) and restore a few for the soon-to-be-executing process (from its kernel stack)
        - The first is when the timer interrupt occurs; in this case, the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.
        - The second is when the OS decides to switch from A to B; in this case, the kernel registers are explicitly saved by the software (i.e., the OS), but this time into memory in the process structure of the process.
        -

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

Scheduling

- Workload assumptions -
  - Each job runs for the same amount of time.
  - All jobs arrive at the same time.
  - Once started, each job runs to completion.
  - All jobs only use the CPU (i.e., they perform no I/O)
  - The run-time of each job is known.
- Scheduling Metric
  - $T_{turnaround} = T_{completion} - T_{arrival}$
- First In, First Out (FIFO)
  - First Come, First Served (FCFS)
  - convoy effect
- Shortest Job First (SJF)
  - If assumption 2 revoked, gone.
  -
- In the old days of batch computing, a number of non-preemptive schedulers were developed; such systems would run each job to completion before considering whether to run a new job.
- Virtually all modern schedulers are preemptive, and quite willing to stop one process from running in order to run another.

- Shortest Time-to-Completion First (STCF)
    - remove assumption 3
    - given our new assumptions, STCF is provably optimal
- $T_{response} = T_{firstrun} - T_{arrival}$
- Round Robin (Good For Response Time)
    - instead of running jobs to completion, RR runs a job for a time slice
    - the length of a time slice must be a multiple of the timer-interrupt period;
    - deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.
    - RR is indeed one of the worst policies if turnaround time is our metric.
- Incorporating I/O
    - remove assumption 4
    - A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is blocked waiting for I/O completion
    - The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state
    - Doing so allows for overlap, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized

How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?

- In our treatment, the MLFQ has a number of distinct queues, each assigned a different priority level. At any given time, a job that is ready to run is on a single queue.
- MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.
    - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
    - Rule 2: If Priority(A) = Priority(B), A & B run in RR.
    - Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
    - Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
    - Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
- The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus quickly alternating between them makes sense .

The Linux Completely Fair Scheduler (CFS)

- To achieve its efficiency goals, CFS aims to spend very little time making scheduling decisions, through both its inherent design and its clever use of data structures well-suited to the task.
- As each process runs, it accumulates vruntime. In the most basic case, each process's vruntime increases at the same rate, in proportion with physical (real) time. When a scheduling decision occurs, CFS will pick the process with the lowest vruntime to run next.
- The first is sched latency. CFS uses this value to determine how long one process should run before considering a switch (effectively determining its time slice but in a dynamic fashion). A typical sched latency value is 48 (milliseconds); CFS divides this value by the number (n) of processes running on the CPU to determine the time slice for a process, and thus ensures that over this period of time, CFS will be completely fair.
- CFS adds another parameter, min granularity, which is usually set to a value like 6 ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overhead.
- CFS utilizes a periodic timer interrupt, which means it can only make decisions at fixed time intervals. This interrupt goes off frequently (e.g., every 1 ms)

CFS maps the nice value of each process to a `weight`, as shown here:

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

These weights allow us to compute the effective time slice of each process (as we did before), but now accounting for their priority differences. The formula used to do so is as follows:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency} \qquad (9.1)$$

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i \qquad (9.2)$$

-
- Keeping the same values in a red-black tree makes most operations more efficient, as depicted in Figure 9.5. Processes are ordered in the tree by vruntime, and most operations (such as insertion and deletion) are logarithmic in time, i.e., O(log n). When n is in the thousands, logarithmic is noticeably more efficient than linear.
- Sleep process: CFS handles this case by altering the vruntime of a job when it wakes up. Specifically, CFS sets the vruntime of that job to the minimum value found in the tree

Mechanism: Address Translation

- With address translation, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address where the desired information is actually located.
- **hardware-based address translation,**
  - is a simple idea referred to as base and bounds; the technique is also referred to as dynamic relocation;
  - need two hardware registers within each CPU: one is called the base register, and the other the bounds (sometimes called a limit register).
  - In the early days, before hardware support arose, some systems performed a crude form of relocation purely via software methods. The basic technique is referred to as static relocation, in which a piece of software known as the loader takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.
  - First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory; in general, hardware support is likely needed for true protection [WL+93]. Another negative is that once placed, it is difficult to later relocate an address space to another location [M65].
  - physical address = virtual address + base
  - A **bounds (or limit) register** ensures that such addresses are within the confines of the address space
  - If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception
  - the part of the processor that helps with address translation **memory management unit (MMU)**
  - to modify the base and bounds registers, These instructions are privileged; only in kernel (or privileged) mode can the registers be modified.

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

  - Figure 15.3: **Dynamic Relocation: Hardware Requirements**

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

- Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**
- This type of waste is usually called internal fragmentation, as the space inside the allocated unit is not all used (i.e., is fragmented) and thus wasted.

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

- Figure 15.5: **Limited Direct Execution (Dynamic Relocation) @ Boot**

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A) to proc-struct(A) (including base/bounds)		
restore regs(B) from proc-struct(B) (including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry in process table		

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

- **Segmentation: Generalized Base/Bounds**
  - we have three logically-different segments: code, stack, and heap. What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.
  - The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address.
  - One common approach, sometimes referred to as an explicit approach, is to chop up the address space into segments based on the top few bits of the virtual address;

- In the implicit approach, the hardware deter-mines the segment by noticing how the address was formed. If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment; if the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap.
- The general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem external fragmentation

- **Paging: Introduction**
  - Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a page.
  - physical memory as an array of fixed-sized slots called page frames
  - To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a page table

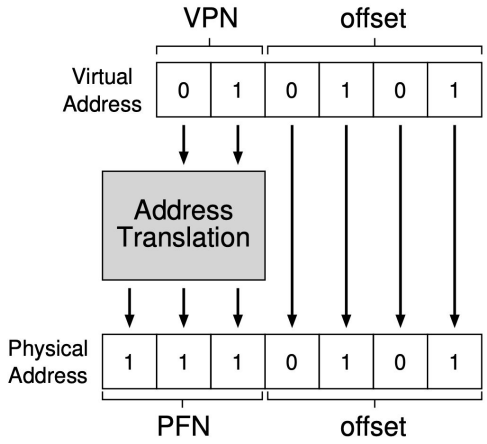


Figure 18.3: **The Address Translation Process**

  -
  - page table entry (PTE) to hold the physical translation plus any other useful stuff
  - page table for each process is in memory somewhere

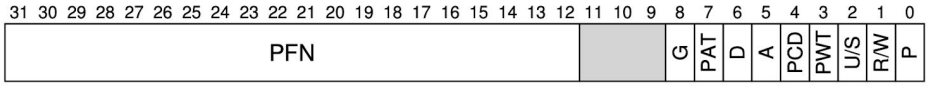


Figure 18.5: **An x86 Page Table Entry (PTE)**

  -
  - protection bits, indicating whether the page could be read from, written to, or executed from
  - A present bit indicates whether this page is in physical memory or on disk (i.e., it has been swapped out)
  - A dirty bit is also common, indicating whether the page has been modified since it was brought into memory

- A reference bit (a.k.a. accessed bit) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory
- It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

- **Translation Lookaside Buffer (TLB)**
  - A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

  -
  - There are usually two types of locality: temporal locality and spatial locality.
  - With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future

Concurrency

- thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.
- There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).
- Instead of a single stack in the address space, there will be one per thread
- The task of transforming your standard single-threaded program into a program that does this sort of work on multiple CPUs is called parallelization
- Threading enables overlap of I/O with other activities within a single program, much like multiprogramming did for processes across programs
  - A critical section is a piece of code that accesses a shared resource, usually a variable or data structure.
  - A race condition (or data race [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the

shared data structure, leading to a surprising (and perhaps undesirable) outcome.
    - An indeterminate program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not deterministic, something we usually expect from computer systems.
    - To avoid these problems, threads should use some kind of mutual exclusion primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.
- `#include int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);`
- There are four arguments: thread, attr, start routine, and arg.
- Commands
    - `pthread create()`
    - `pthread_attr_init()`
    - `pthread_join()`
    - `pthread_mutex_lock`
    - `pthread_mutex_unlock`
    - `pthread_mutex_init`
    - `pthread_mutex_destroy`
- The `trylock` version returns failure if the lock is already held; the `timedlock` version of acquiring a lock returns after a timeout or after acquiring the lock, whichever happens first. Thus, the timedlock with a timeout of zero degenerates to the trylock case
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- Evaluating Locks
    - mutual exclusion
    - fairness
    - performance
- The simplest bit of hardware support to understand is known as a test-and-set); on x86 it is the locked version of the atomic exchange (xchg).
- **Peterson's algorithm (2 threads)**

```
int flag[2];
int turn;
void init() {
     // indicate you intend to hold the lock w/ 'flag'
     flag[0] = flag[1] = 0;
     // whose turn is it? (thread 0 or 1)
     turn = 0;
}
void lock() {
     // 'self' is the thread ID of caller
```

```
      flag[self] = 1;
      // make it other thread's turn
      turn = 1 - self;
      while ((flag[1-self] == 1) && (turn == 1 - self))
      ; // spin-wait while it's not your turn
}
void unlock() {
      // simply undo your intent
      flag[self] = 0;
}
```

- Another hardware primitive that some systems provide is known as the compare-and-swap instruction (as it is called on SPARC, for example), or compare-and-exchange (as it called on x86).
- The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the storeconditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.

```
int LoadLinked(int *ptr) {
      return *ptr;
}

int StoreConditional(int *ptr, int value) {
      if (no update to *ptr since LoadLinked to this address) {
            *ptr = value;
            return 1; // success!
      } else {
            return 0; // failed to update
      }
}
void lock(lock_t *lock) {
      while (1) {
      while (LoadLinked(&lock->flag) == 1)
      // spin until it's zero
      if (StoreConditional(&lock->flag, 1) == 1)
      return; // if set-it-to-1 was a success: all done
      // otherwise: try it all over again
      }
}

void unlock(lock_t *lock) {
      lock->flag = 0;
}
```

- One final hardware primitive is the fetch-and-add instruction, which atomically increments a value while returning the old value at a particular address

```
int FetchAndAdd(int *ptr) {
     int old = *ptr;
     *ptr = old + 1;
     return old;
}
typedef struct __lock_t {
     int ticket;
     int turn;
} lock_t;

void lock_init(lock_t *lock) {
     lock->ticket = 0;
     lock->turn = 0;
}

void lock(lock_t *lock) {
     int myturn = FetchAndAdd(&lock->ticket);
     while (lock->turn != myturn)
          ; // spin
}

void unlock(lock_t *lock) {
     lock->turn = lock->turn + 1;
}
```

- Can yield simply. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section. We clearly will need an approach that addresses this problem directly. (starvation)

FileSys

- Access Paths: Reading and Writing
    - open("/foo/bar", O RDONLY)
    - The file system must traverse the pathname and thus locate the desired inode.
    - the first thing the FS will read from disk is the inode of the root directory
    - Usually, we find the i-number of a file or directory in its parent directory
    - In most UNIX file systems, the root inode number is 2. Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).
    - The FS will thus use these on-disk pointers to read through the directory, in this case looking for an entry for foo. By reading in one or more directory data blocks,

it will find the entry for foo; once found, the FS will also have found the inode number of foo (say it is 44) which it will need next.

- READS DON'T ACCESS ALLOCATION STRUCTURES (bmap, imap)
- The final step of open() is to read bar's inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.
- At some point, the file will be closed. There is much less work to be done here; clearly, the file descriptor should be deallocated, but for now, that is all the FS really needs to do. No disk I/Os take place.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**			**read**				
				read			**read**			
					read					
read()					**read**			**read**		
					write					
read()					**read**				**read**	
					write					
read()					**read**					**read**
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

-
- Also note that the amount of I/O generated by the open is proportional to the length of the pathname. For each additional directory in the path, we have to read its inode as well as its data. Making this worse would be the presence of large directories; here, we only have to read one block to get the contents of a directory, whereas with a large directory, we might have to read many data blocks to find the desired entry

- Writing A File To Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read	read write write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

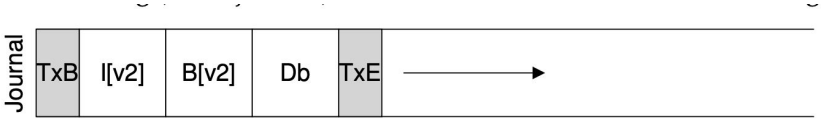
- ○
- ○ each write to a file logically generates five I/Os: one to read the data bitmap (which is then updated to mark the newly-allocated block as used), one to write the bitmap (to reflect its new state to disk), two more to read and then write the inode (which is updated with the new block's location), and finally one to write the actual block itself.
- ○ You can see how much work it is to create the file: 10 I/Os in this case, to walk the pathname and then finally create the file. You can also see that each allocating write costs 5 I/Os: a pair to read and update the inode, another pair to read and update the data bitmap, and then finally the write of the data itself.

- Caching and Buffering
  - ○ Now imagine the file open example with caching. The first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.
  - ○ For the reasons above, most modern file systems buffer writes in memory for anywhere between five and thirty seconds, representing yet another trade-off: if the system crashes before the updates have been propagated to disk, the updates are lost; however, by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes
  - ○ Some applications (such as databases) don't enjoy this trade-off. Thus, to avoid unexpected data loss due to write buffering, they simply force writes to disk, by calling fsync(), by using direct I/O interfaces that work around the cache, or by using the raw disk interface and avoiding the file system altogether

    - blk = (inumber * sizeof(inode_t)) / blockSize;
    - sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
- FSCK
    - One major challenge faced by a file system is how to update persistent data structures despite the presence of a power loss or system crash.
    - When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block and record the new larger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.
    - Note that these writes usually don't happen immediately when the user issues a write() system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the page cache or buffer cache) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk.
    - Crash Scenarios
        - Just the data block (Db) is written to disk. In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency
        - Just the updated inode (I[v2]) is written to disk. we will read garbage data from the disk. we have a new problem, which we call a file-system inconsistency. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has.
        - Just the updated bitmap (B[v2]) is written to disk. if left unresolved, this write would result in a space leak, as block 5 would never be used by the file system
        - The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db). In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
        - The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]). In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.
        - The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]). In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.
    - The File System Checker
        - Here is a basic summary of what fsck does:

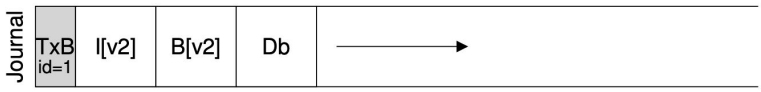
- **Superblock:** fsck first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated.
- **Free blocks:** Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.
- **Inode state:** Each inode is checked for corruption or other problems. For example, fsck makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by fsck; the inode bitmap is correspondingly updated.
- **Inode links:** fsck also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, fsck scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory
- **Duplicates:** fsck also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks**: A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered “bad” if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, fsck can’t do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** fsck does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, fsck performs additional integrity checks on the contents of each directory, making sure that “.” and “..” are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy

- Journaling
  - When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the “write ahead” part, and we write it to a structure that we organize as a “log”; hence, write-ahead logging.
  - Say we have our canonical update again, where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal).



  -
  - The final block (TxE) is a marker of the end of this transaction, and will also contain the TID
    - **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
    - **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be committed.
    - **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.
    - **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):



When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:

    -
  - To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to

write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

- Metadata Journaling
  - **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
  - **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
  - **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now committed.
  - **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
  - **Free:** Later, mark the transaction free in journal superblock.

References

1. *Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, August 2018*

Additional Lecture Notes

Signals

- In many scenarios, either the OS or some other processes may want to send some notifications to a particular user process. For example, if a process is executing an infinite loop because of programming errors, the user may want to kill the process, the OS may want to notify the process if it performs a division-by-zero so that the process can perform cleanup activities before exiting, a parent process may want a notification from the OS when one of its child processes terminates.
- UNIX OSs provide the signal system call API to enable user processes register notification handlers to different events. The signal system call allows a user process to register handlers for different events (signals). The signature of signal system call is as follows, **`signal`** `(signum, sighandler)`
- signum is an integer specifying the signal (event) for which the user process wants notification from the OS. Some examples of signal numbers are, SIGINT, SIGHUP, SIGALRM, SIGCHLD etc. sighandler is the function pointer of the handler which is invoked when the signal event occurs. For example, if **`signal`**`(SIGINT, sigint_handler)` is invoked from a process, the sigint_handler function is executed every time CTRL+C is pressed on the console where the process is executing. The value of sighandler can be SIG_DFL to specify default handler (inside OS) or SIG_IGN to ignore the signal. **Note that, for some signals (like SIGKILL, SIGQUIT), a user process is not allowed to ignore or register a custom handler.**
- A signal can be sent from a process to another process using the kill system call. **`kill`**`(pid, SIGSEGV)` sends the `SEGV (segfault)` signal to the process with `PID=pid`. If pid is zero, the signal will be delivered to all processes in the calling process group. Note that, all child processes belong to the same process group of the parent unless the child executes `setpgrp()` system call to set its own process group.
- The OS maintains a per-process vector of handlers corresponding to all signals in the process control block (PCB). The value of handler can be one of the following: Default handlers, no handler (ignore) or custom handler (registered by the process using `signal()` system call). Further, the OS also maintains a pending signal bit vector where each bit corresponds to a signal which is either set by the OS to indicate an event like SEGFAULT or by another process through the `kill()` system call.
- Signals can be delivered in a synchronous manner when the event occurs because of any misbehavior of the executing process. For example, if a process causes an exception (by performing division by zero), the signal may be delivered when returning to the user space after the exception handling in the OS. In many other cases, if the process is executing in user mode when the signal event occurs, the signal has to be delivered in a deferred (asynchronous) manner. For example, if process A wants to send a signal to process B which is executing in user mode, the signal pending bit is set to one (in the PCB of B) and the signal is sent to B when the OS gains the execution control.
- The OS can invoke the signal handler of a process when it is returning from an exception handler, system call handler or an interrupt handler. Therefore, the maximum time the

signal remains undelivered depends on two factors---(i) the time till the OS gains control from the process and, (ii) the time till the process gets scheduled again.

- In case of a custom signal handlers, the OS returns to the user process resuming execution of the handler function after which the control comes back to the point where the process was executing before. To realize this behavior, the OS must mimic a function call in the user space. This requires careful manipulation of the user space stack and the instruction pointers by the OS. One possible design can be to create a new stack frame with return address set to the original user execution point (value of RIP before the entry into OS through a syscall, exception or an interrupt). On X86 systems, this is possible as the user process execution details (user RSP, RIP etc.) are pushed to the OS stack and the OS can access and appropriately augment them.
- An important design consideration for the OS is to design signal delivery mechanisms in presence of nested signals. More specifically, if a signal event occurs while one instance of the same signal is being handled, what should the OS do? If the signals are delivered during signal handling, the OS needs to carefully manage the user stack and further, the user stack may overflow in case of multiple nested invocation of signal handlers. Another alternative can be to disable signals during the execution of signal handlers and the user process re-registers the signal after handling the signal. There are multiple issues with this approach---lost signals, burden on the programmer to re-enable etc.
- Modern days OSs like Linux, insert system calls in the execution flow in a user transparent manner where the signal gets re-enabled through a special system call like sigreturn( ) when the signal handler finishes. This can be done using techniques similar to signal handler call mimicking through user stack manipulations or creating a temporary stack (a.k.a. Signal stack).
- Signal handlers are inherited by the child processes. However, **the pending signals (if any) bit vector is all cleared for the child and are not delivered to the child**. This makes sense as the pending signals are meant for the parent because they are set before the child creation.

Address space (examples and class notes)

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

extern char etext, edata, end; /*see man(3) end*/

int main()
{

   printf("End of text %p\n", &etext);
   printf("End of initialized data %p\n", &edata);
   printf("End of uninitialized data %p\n", &end);
   return 0;

 }
```

The above program prints the load time end address of different segments. Output on my machine is

```
End of text 0x4005fd

End of initialized data 0x601040

End of uninitialized data 0x601048
```

Note that, on your machines, these addresses may be different. The addresses are hardcoded by the compiler as we see in the object dump using gdb (annotated and highlighted below)

```
    0x000000000040052d <+0>: push   %rbp
   0x000000000040052e <+1>: mov    %rsp,%rbp
   0x0000000000400531 <+4>: mov    $0x4005fd,%esi
   0x0000000000400536 <+9>: mov    $0x400604,%edi  // 0x400604 contains the format string "End
of text %d\n"
   0x000000000040053b <+14>: mov    $0x0,%eax
   0x0000000000400540 <+19>: callq  0x400410 <printf@plt>
   0x0000000000400545 <+24>: mov    $0x601040,%esi
   0x000000000040054a <+29>: mov    $0x400614,%edi
   0x000000000040054f <+34>: mov    $0x0,%eax
   0x0000000000400554 <+39>: callq  0x400410 <printf@plt>
   0x0000000000400559 <+44>: mov    $0x601048,%esi
   0x000000000040055e <+49>: mov    $0x400630,%edi
   0x0000000000400563 <+54>: mov    $0x0,%eax
   0x0000000000400568 <+59>: callq  0x400410 <printf@plt>
   0x000000000040056d <+64>: mov    $0x0,%eax
   0x0000000000400572 <+69>: pop    %rbp
   0x0000000000400573 <+70>: retq
```

In 64-bit X86, parameters are passed by registers (if possible). Now let us see how the variables are allocated in different segments using the following example.

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

extern char etext, edata, end; /*see man(3) end*/

int global_x = 5;  // Initialized data
char gchar[32];    // Uninitialized data

int main()
{
   int x = 5;   // stack
   char *ptr, *ptr1; // stack
   ptr = malloc(1024);  // heap

   printf("==========Text=========\n");
   printf("Address of main = %p\n", &main);
   printf("End of text %p\n", &etext);

   printf("==========Data(initialized)=========\n");
   printf("Address of global_x = %p\n", &global_x);
   printf("End of initialized data %p\n", &edata);

   printf("==========Data(uninitialized)=========\n");
```

```
   printf("Address of gchar= %p\n", gchar);
   printf("End of uninitialized data %p\n", &end);

   printf("==========Stack=========\n");
   printf("Address of x = %p\n", &x);
   printf("Address of ptr = %p\n", &ptr);

   printf("==========Heap=========\n");

   printf("ptr = %p\n", ptr);
   ptr1 = malloc(1024);
   printf("ptr1 = %p\n", ptr1);

   exit(0);

 }
```

The above program prints the addresses of variables allocated from different segments of the address space. The output of this program on my machine is:

```
==========Text=========
Address of main = 0x40060d
End of text 0x4007cd
==========Data(initialized)=========
Address of global_x = 0x601058
End of initialized data 0x60105c
==========Data(uninitialized)=========
Address of gchar= 0x601080
End of uninitialized data 0x6010a0
==========Stack=========
Address of x = 0x7fffffffe53c
Address of ptr = 0x7fffffffe540
==========Heap=========
ptr = 0x602010
ptr1 = 0x602420
```

Note that, the address of different variables lie within the span of the respective segments. What about the allocations done using `malloc()`? Let us see what syscalls are triggered to satisfy the malloc (using strace). The relevant output lines on my machine using 'strace ./a.out' are quoted below.

```
brk(0)                                  = 0x602000    // current end of BSS (at
page boundary i.e., next nearest 4KB aligned address).
                                                                  // see MAN
page of  brk and its behaviour in Linux
brk(0x623000)                           = 0x623000 // New end of BSS
```

Note that, for the second malloc call (for ptr1 towards the end of main()), no system calls are passed.

Important Note: You may get different address for the ptr (not just after the end of BSS) if you execute it multiple times. For predictable address on Linux, make sure you have disabled VA randomization using the following command. VM randomization, a.k.a address space layout randomization (ASLR) is a security feature in kernel to provide randomness in address space allocations.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

If we change the size of allocation by passing a different size (e.g., 1024*1024*4 = 4MB), the allocation method may change.

HW(1): Try to change the allocation size and observe the behavior

HW(2): Change the above program to allocate 1GB (using malloc) without using it. Observe the change in memory usage in the system using "free -m" command during the execution of the program.

The below given program shows an example of using sbrk( ).

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

extern char etext, edata, end; /*see man(3) end*/

int main()
{
   printf("End of text %p\n", &etext);
   printf("End of initialized data %p\n", &edata);
   printf("End of uninitialized data %p\n", &end);
   printf("Current heap start = %p\n", sbrk(0));
   if(sbrk(4096 * 1024) == (void *)-1){
        printf("sbrk failed\n");
   }
   printf("Heap start after expand = %p\n", sbrk(0));

 }
```

Output on my system is:

```
End of text 0x4006dd

End of initialized data 0x601050

End of uninitialized data 0x601058

Current heap start = 0x602000

Heap start after expand = 0xa02000
```