Operating Systems

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison

This note is a reading note of the book: Operating Systems: Three Easy Pieces (OSTEP) v1.01 by Prof. Remzi Arpaci-Dusseau and Prof. Andrea Arpaci-Dusseau. Figures included in this note are from this book unless otherwise stated.

Introduction to OS

An **Operating System (OS)** is a body of software sitting in between *software applications* and a *Von Neumann* computer architecture. An OS connects applications to physical hardware. It makes **abstractions** of the underlying hardware and provides an easy-to-use *interface* for running portable software on physical hardware.

An OS does this through three general techniques:

1. **Virtualization**: taking a physical resource (processor, memory, storage, ...) and transforms it into a more general, portable, and easy-to-use virtual interface for user applications to use
2. **Concurrency**: acting as a resource manager which supports multiple user applications to run concurrently and coordinates among running applications, ensuring correct, fair, and efficient sharing of resources
3. **Persistence**: data can be easily lost on volatile devices such as DRAM. An OS allows users to talk to external devices - including persistent storage drives - through *Input/Output* (I/O)

> Abstraction is a great idea in both computer architecture and operating systems. It hides implementation complexity about the underlying layer and exposes a unified model of how to use the underlying layer to the upper layer. Check out the first section of this note.
>
> In the layers of abstractions, we call the operations supported by the lower-layer as **mechanisms**, and the algorithms/decisions made by the higher-layer on how to use the mechanisms to achieve a goal as **policies** (or *disciplines*).

A modern operating system also pursues some other goals apart from the above stated. They are:

- *Performance*: minimize the overheads brought by the OS
- *Scalability*: with multiprocessors, speed up concurrent accesses
- *Security*: protect against bad behavior; provide *isolation*
- *Reliability*: properly recover on fail-stops
- *Connectivity*: networking support; connect with the Internet
- *Energy-efficiency*, *Mobility*, ...

> Generally, the most essential part of an OS is the **kernel**, which is the collection of code that implements the above-mentioned core functionalities. The kernel is a piece of static code (hence not a dynamic running entity). A monolithic OS consists of the kernel and some upper-level system applications that provide a more friendly UI.

Virtualizing the CPU: Processes & Scheduling

One of the most important abstractions an OS provides is the **process**: a running instance of a program. We typically want to run many processes at the same time (e.g., a desktop environment, several browser windows, a music player, and a task monitor), more than the number of available CPU cores (and probably other physical resources as well). The OS must be able to *virtualize* a physical resource and let multiple processes share the limited resource. This section focuses on sharing the CPU, which is the most fundamental resource required to kick off any process.

Abstraction of Process

A process is simply an *instance* running on a processor (the dynamic instance, doing actual work) of a piece of *program* (the static code + data, residing on persistent storage). There can be multiple processes running the same piece of program code.

Machine State

Running a process instance of a program requires the OS to remember the *machine state* of the process, which typically consists of the following information:

- **Address space**: memory space that a process can address, typically also virtualized, which contains at least:
    - *Code*: compiled machine code of the program
    - *Data*: any initial static data/space the program needs
    - *Stack*: space reserved for the run-time function *stack* of the process
    - *Heap*: space reserved for any new run-time data
- **Registers context**: CPU registers' values; particularly special ones include:
    - *Program counter* (PC, or *instruction pointer*): which instruction of the program to execute next
    - *Stack pointer* (SP) & *Frame pointer* (FP): for managing the function stack
- **I/O information**: states related to storage or network, for example:
    - List of currently open files (say in the form of *file descriptors*)

Process Status

A process can be in one of the following states at any given time:

- (optional) *Initial*: being created and hasn't finished initialization yet
- *Ready*: is ready to be scheduled onto a CPU to run, but not scheduled at this moment
- *Running*: scheduled on a CPU and executing instructions
- *Blocked*: waiting for some event to happen, e.g., waiting for disk I/O completion or waiting for another process to finish, hence not ready to be scheduled at this moment
- (optional) *Terminated*: has exited/been killed but its information data structures have not been cleaned up yet

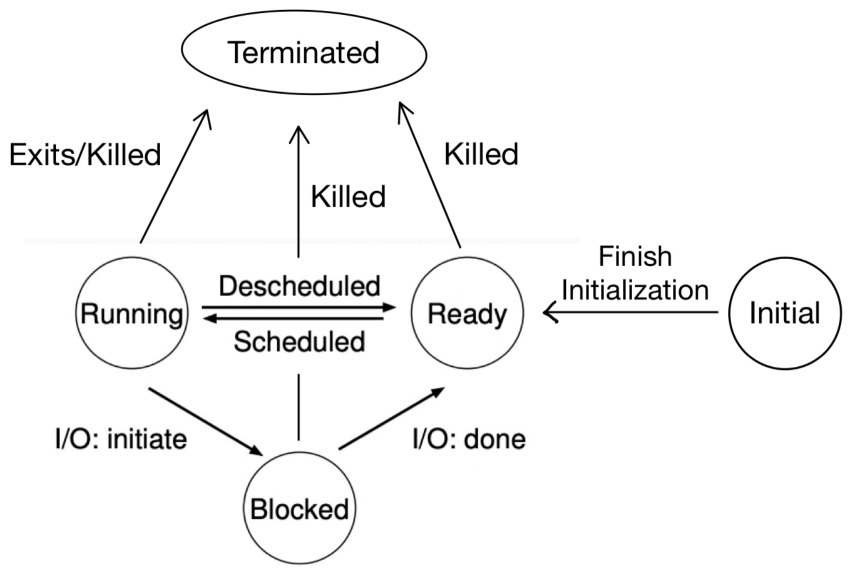


Figure 4.2: **Process: State Transitions** (extended)

Process Data Structures

The OS must have some data structures to hold the information of each process. We call the metadata structure of a process the **process control block** (PCB, or *process descriptor*). This structure must include the machine state of the process, the status of the process, and any other necessary information related to the process. For example, the xv6 OS has the following PCB struct:

```c
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

/** The PCB structure. */
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If !zero, sleeping on chan
    int killed;                 // If !zero, has been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Register values context
    struct trapframe *tf;       // Trap frame of current interrupt
};
```

The collection of PCBs is the **process list** (or *task list*), the first essential data structure we meet in an OS. It can be implemented as just a fixed-sized array of PCB slots as in xv6, but can also take other forms such as a linked list or a hash table.

Process APIs

These interfaces are available on any modern OS to enable processes:

- `create` - *initialize* the above mentioned states, *load* the program from persistent storage in some *executable format*, and get the process running at the entry point of its code; the process then runs until completion and exits (, or possibly runs indefinitely)

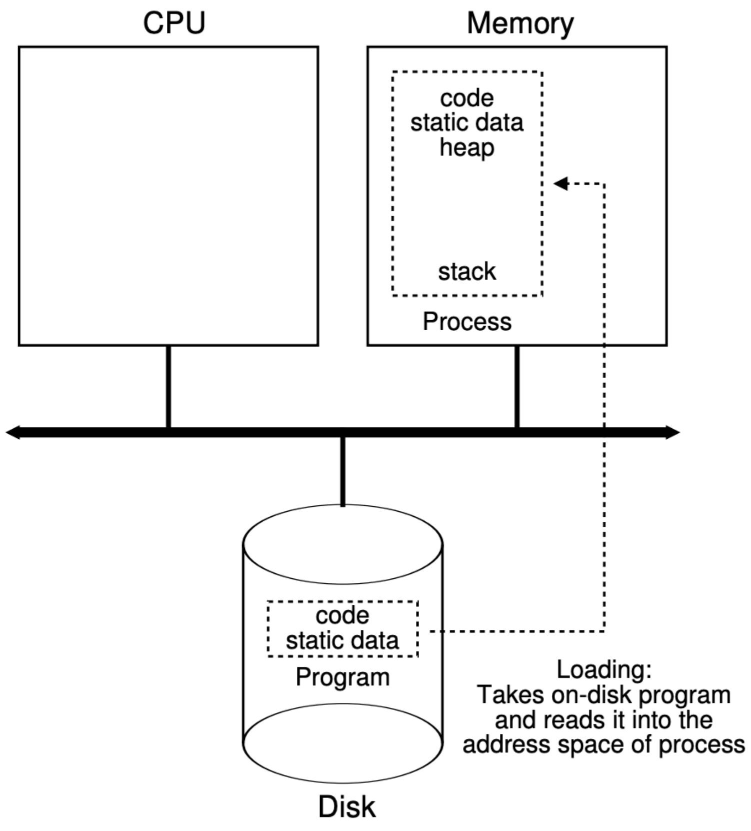


Figure 4.1: **Loading: From Program To Process**

- `destroy` - forcefully destroy (kill, halt) a process in the middle of its execution
- `wait` - let a process wait for the termination of another process
- `status` - retrieve the status information of a process
- other control signals like suspending & resuming, ...

> *Application programming interface* (API) means the set of interfaces a lower-layer system/library provides to upper-layer programs. The APIs that an OS provides to user programs are called **system calls** (*syscalls*). Everything a user program wants to do that might require system-level privileges, such as creating processes, accessing shared resources, and communicating with other processes, are typically done through invoking system calls.

In UNIX systems, process creation is done through a pair of `fork()` + `exec()` syscalls.

- Initialization of a process is done through `fork()`ing an existing process: create an exact duplicate
  - The original process is the *parent* process and the created duplicate is the *child* process
  - There is a special `init` process created by the OS at the end of the booting process, and several basic user processes, e.g., the command-line shell or the desktop GUI, are forked from the `init` processes. They then fork other processes, for example when the user opens a calculator from the shell, forming a tree of processes
- Executable loading is done through `exec()`ing an executable file: replace the code and data section with that executable and start executing it from its entry

See Chapter 5, Figure 5.3 for a fork+exec+wait example, Section 5.4 for an introduction to UNIX shell terminologies, and Section 5.5. for control signals handling.

Time-Sharing of the CPU

To virtualize the CPU and coordinate multiple processes, the virtualization must be both *performant* (no excessive overhead) and *controlled* (OS controls which one runs at which time; bad processes cannot simply run forever and take over the machine). OS balances these two goals with the following two techniques:

- **(Limited) Direct Execution** (LDE): just run the user program directly on the CPU; OS is not simulating processor hardware; but the OS must be able to re-gain control in some way to do the coordination
- **Time-Sharing**: divide time into small slots, schedule a process to run for a few slots and switch to another one, constantly switching back and forth among ready processes

The OS needs to solve several problems to enable the combination of these two techniques.

Privilege Modes

If we just let a process run all possible instructions directly, it will have dominant control over the machine. Hence, the processor lists a set of sensitive instructions as *privileged* instructions, which can only be run in high privilege mode. A user process normally runs in **user mode**, with restricted permissions. When it invokes a syscall (mentioned in the section above), it switches to **kernel mode** to execute the registered syscall handler containing privileged instructions. After the handler has done its work, it returns the process back to user mode.

Examples of privileged instructions in x86 (can only be called in "ring-0" mode, as opposed least-privileged "ring-3" mode) include:

- `HALT` - stop execution
- I/O port instructions
- Turning off interrupts
- ...

The switching between modes is enabled through a mechanism called **trap**. The special trap instruction simultaneously jumps into somewhere in kernel mode (identified by a *trap number*) and raises the privilege level to kernel mode. This also includes changing the stack pointer to the *kernel stack* reserved for this process, and saving the process's user registers into the kernel stack.

To let the hardware know where is the corresponding handler code for a given trap number, the OS registers the address of a *trap table*: trap no. $\rightarrow$ trap handler addresses into a special hardware register.

This trapping mechanism provides protection since a user process cannot do arbitrary system-level actions, but rather must request a particular action via a number.

> Trapping into kernel from software through a syscall is sometimes called a *software interrupt*.
>
> We often use *trap* to name an active syscall made by the program. We use *exception* (or *fault*) to name a passive fall-through into kernel if the program misbehaves (e.g., tries to directly call a privileged instruction) or encounters an error (e.g., runs out-of-memory, OOM).

Hardware Interrupts

Another big issue is that the OS must be able to re-gain control in the middle of the user-level execution of a process, and achieve a switch between processes. We need help from a special hardware component - the *timer*.

A process can trap/fault into kernel, so does a hardware device. The behavior of a hardware device sending a signal to the processor to pause it current user code execution and to run a specified handler is called an **interrupt**. The handler is often called an *interrupt service routine* (ISR). Similar to traps, there is an *interrupt table* from interrupting device no. $\rightarrow$ interrupt handler addresses (sometimes the software trap table is just embedded in the interrupt table).

One particularly important type of interrupt is the **timer interrupt**, issued by the timer every configured interval (e.g., 10ms). The ISR for timer interrupt is the place where the OS re-gains control on deciding which process to schedule for the next time slot.

Examples of other hardware interrupts include:

- Keystroke of a specific key from a keyboard device
- Mouse movement/click; other *peripheral devices*
- ...

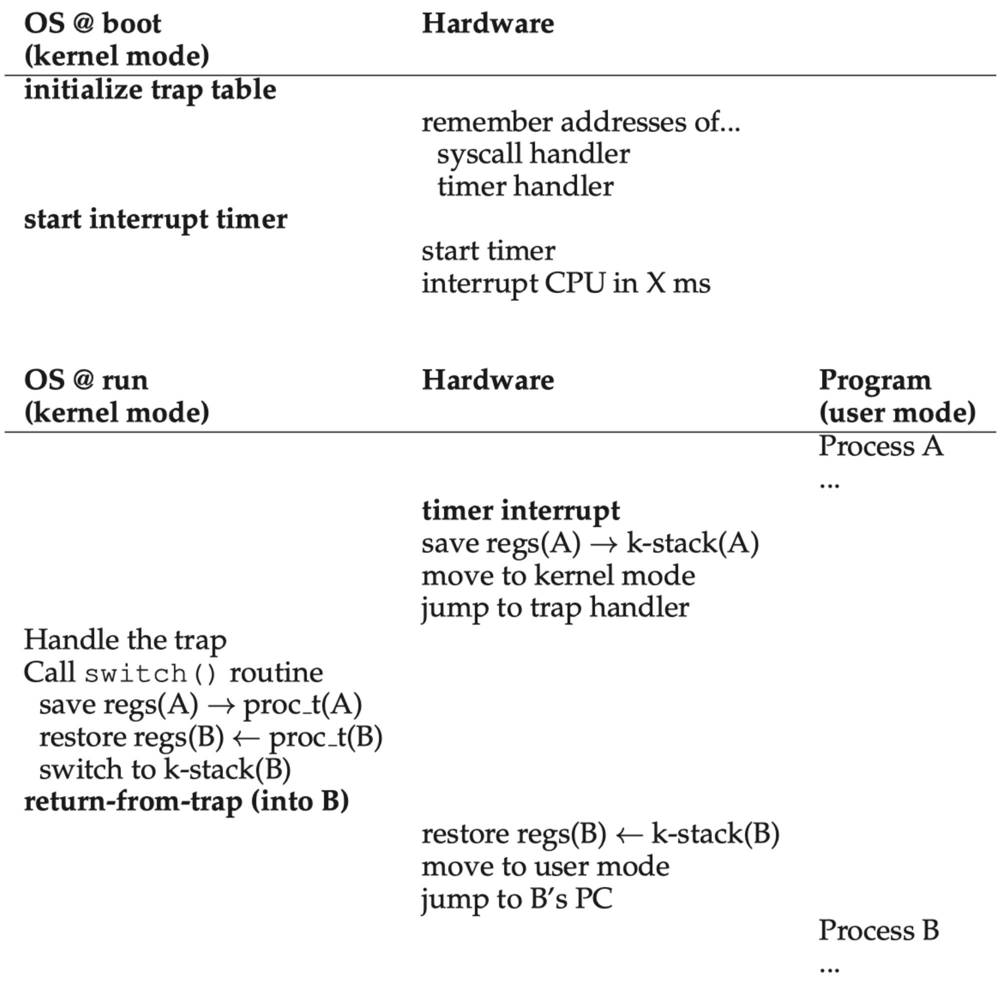
> Interrupts need to be *disabled* (meaning not obeying incoming interrupts) during the handling of a trap/interrupt, so that every interrupt is handled to its completion.
>
> Also see double fault and triple fault for how the architecture reacts to faults happened inside trap handlers/ISRs.

Context Switch & Scheduler

We have defined the context of a process as the set of important registers values. Switching from process A to process B is a **context switch** procedure:

1. Save the current registers values (the context of A) into A's PCB
2. Restore the saved register values (the context) of B from B's PCB
3. Jump to where B was left off - most likely B is in its kernel stack right after the switch, because the last time B was scheduled out, it must be in the scheduler routine

The timer interrupt ISR typically calls the **scheduler**: a routine that chooses which process to run for the next time slot, possibly using its knowledge of what were scheduled in the history, based on a scheduling policy. It then performs a context switch to switch to the target process. The entire picture looks like:



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)**

> Note the difference between *user registers* (those at user-level execution of the program) and *kernel registers* (those actually stored in the context). At a timer interrupt, the user registers are already stored into the process's kernel stack and the registers have been switched to the set for kernel execution (PC pointing to the handler code, SP pointing to kernel stack, etc.). Hence, the context saved for a process is actually the set of its kernel registers, and the user registers are recovered by the "return-from-trap" operation.

Context switch is not free and there is observable *overhead* (saving/restoring registers, making caches/TLBs/branch predictors cold, etc.). We should not ignore this overhead when developing scheduling policies.

Scheduling Policies

With the trap/interrupt mechanism and the context switch mechanism in hand, it is now time to develop scheduling policies for the scheduler to decide which process runs next. Determining the *workload* - here the processes running in the system and their characteristics - is critical to building policies. A policy can be more find-tuned if we know more about the workload.

Basic Policies

Let's first (unrealistically) assume that our workloads are:

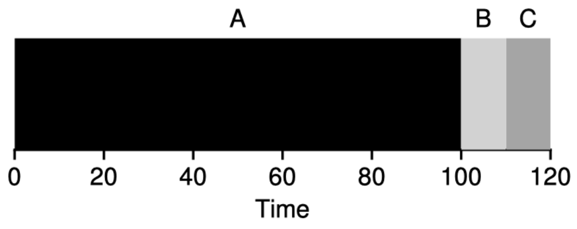
- All jobs only use the CPU (so a single resource w/o blocking) and perform no I/O
- The run-time of each job is known

We look at these metrics to compare across policies:

- **Turnaround time** $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$ for each job, then averaged over jobs; if with the assumption of "jobs arrive at the same time", T_{arrival} will be 0 for everyone
- **Response time** $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$ for each job; it measures how "promptive" an interactive job (say the shell) would be
- **Fairness**: we often need to *tradeoff* between turnaround time and fairness

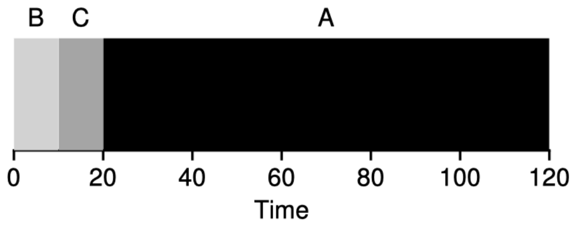
Some basic, classic policies are:

- **First-In-First-Out** (FIFO, or *First-Come-First-Serve*, FCFS)



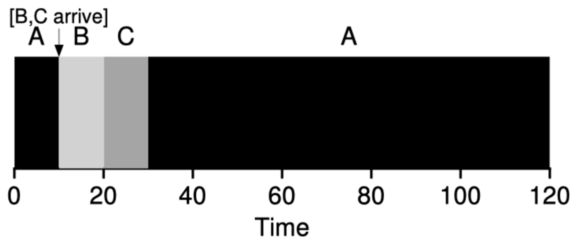
  FIFO is intuitive but can often leads to the *convoy effect*: where a number of relatively short jobs get queued after a long job, yielding poor overall turnaround time. FIFO may work well in some cases such as caching, but not for CPU scheduling.

- **Shortest-Job-First** (SJF)



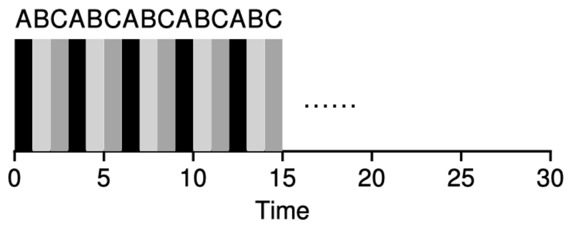
  SJF is optimal on turnaround time if all jobs arrive at the same time, each job runs until completion, and their durations are known; however, in reality, this is rarely the case, and suppose A comes a little bit earlier than B & C in the above example, it encounters the same problem as in FIFO.

- **Shortest-Time-to-Completion-First** (STCF, or *Preemptive-Shortest-Job-First*, PSJF)



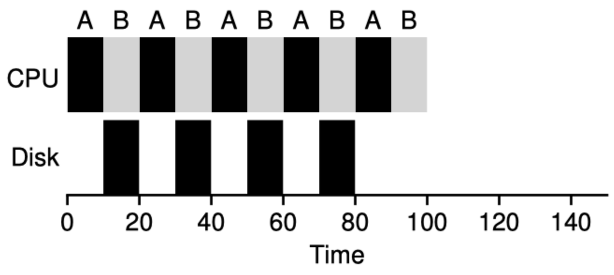
  STCF is a *preemptive* policy - it can **preempt** a job in the middle if another one with shorter TtoC arrives. (Accordingly, FIFO and SJF schedulers are *non-preemptive*.) STCF is optimal given our current assumption and the turnaround time metric.

- **Round-Robin** (RR)



Above-mentioned policies are not really taking advantage of the time-sharing mechanisms. RR divides time into small slots (*scheduling quantum*) and then switches to the next job in the run queue in a determined order. RR is a **time-slicing** scheduling policy. The time slice length should be chosen carefully: short enough to be responsive, and long enough to *amortize* the cost of context switches.

RR is quite fair and responsive, but one of the worst on turnaround time. With RR, we can also take I/O into consideration: when job A blocks itself on doing a disk I/O, the scheduler schedules B for the next slot, *overlapping* these two jobs.



Multi-Level Feedback Queue (MLFQ)

The biggest problem of the above basic policies is that they assume job durations are known beforehand (*priori* knowledge). However, this is hardly true in any real systems. Good scheduling policies tend to learn something from jobs' past behavior (*history*) and make decisions accordingly. One of the best-known approaches is **Multi-Level Feedback Queue** (MLFQ). It tries to trade-off between optimizing turnaround time and minimizing response time. In other words, it well mixes *interactive* or *I/O-intensive* jobs with long-running *CPU-intensive* jobs.

MLFQ has a number of distinct *queues*, each assigned a different **priority** level. A job is in a single queue at any given time.

- If $\text{priority}(A) > \text{priority}(B)$, i.e., A is in a higher level than B, A runs;
- If $\text{priority}(A) = \text{priority}(B)$, i.e., for jobs in the same level, they run in RR.

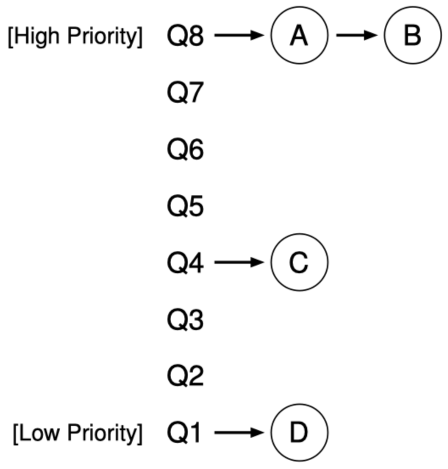


Figure 8.1: **MLFQ Example**

The key lies in how MLFQ dynamically sets priorities for jobs. MLFQ first assumes that a job might be short-running & interactive, hence giving it highest priority. Then, every time it runs till the end of a time slice without blocking itself on e.g. I/O, we reduce its priority level by 1, moving it to the queue one level down.

- When a job enters, it is placed at the highest priority (the topmost queue);
- If a job uses up an entire time slice without relinquishing the CPU, it moves down one queue;
- If a job gives up the CPU before the time slice is up, it stays in the same priority level queue.

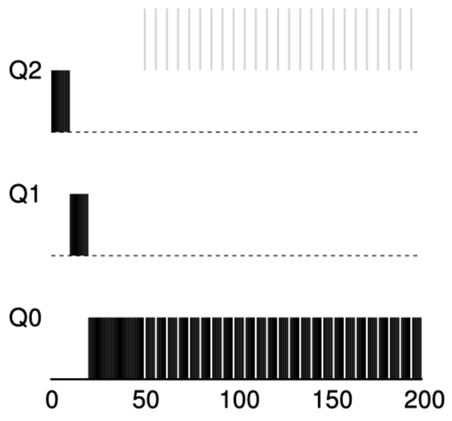


Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

There are still a few problems. There might be *starvation* if there are quite a few interactive jobs that a long-running job might have no chance to run. A program could *game* the scheduler by issuing an I/O at the very end of every time slice. The behavior of a job might also change over time. Hence, we add several rules:

- *Priority boost*: after some time period S, move all the jobs in the system to the topmost queue;
- *Gaming tolerance*: once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), it moves down one queue.

Smarter MLFQ implementations tune these parameters instead of setting them as a default constant: adjusting #priority levels (#queues), lower levels have longer time slice quanta, using math formula (e.g., decaying) to calculate quanta and allotment, etc. OS implementations, such as FreeBSD, Solaris, and Windows NT, use some form of MLFQ as the base scheduler.

Lottery Scheduling

We also examine a different type of scheduler known as **fair-share** scheduler. Instead of minimizing turnaround time, it tries to guarantee that each job obtain a certain percentage of CPU time - a *proportional share*. **Lottery Scheduling** is a good example of such scheduler, see the paper.

Lottery scheduling assigns each job a number of *tickets*. At each scheduling decision point, it uses *randomness* to decide who wins the next slot. Say A has 75 tickets, B has 25, and C has 100, so the total amount of tickets is 200. The systems picks a random number between 0~199 and walks a running sum: if 75 > num, A wins, else if 75+25=100 > num > 75, B wins, otherwise C wins.

There are a few extra ticket mechanisms that might be useful:

- *Ticket currency*: a group/user can allocate tickets among their own child jobs in whatever scale - they form a hierarchy of currencies and the eventual proportions are multiplied (Figure 3 of the paper)
- *Ticket transfer*: a process can temporarily transfer tickets to another process to boost its share, e.g., when a client sends a request to a server process and wants it to finish the request quickly
- *Ticket inflation*: in a *cooperative* (*trusted*) scenario, a process can simply inflate its number of tickets to reflect the need of more CPU share, without even communicating with any shared state in kernel

> *Stride scheduling* avoids using randomness but instead divides the #tickets by a large number to get the *stride* value of each process. Whenever a process finishes a slot, increment its *pass* value by its stride. Pick the process with smallest pass value to run next. The strength of lottery scheduling (using randomness) is that it does not need any global state.

Completely Fair Scheduler (CFS)

Linux adopts a scalable, weighted-RR, fair-share scheduler named the **Completely Fair Scheduler** (CFS), see here. It tries to be efficient and scalable by spending very little time making scheduling decisions. CFS accounts *virtual runtime* (`vruntime`) for every job.

- Each job's `vruntime` increases at the same rate with physical time; always pick the process with the lowest `vruntime`;
- Parameter `sched_latency` (e.g., 48ms) decides the length of a whole round; CFS divides this value by the number of processes to determine the time slice length for each process;
- CFS never assigns time slice lengths shorter than parameter `min_granularity`, so it won't go wrong if there are too many processes;
- Each process has a *nice* value from -20 to +19, positive niceness implies lower priority (so smaller weight); there is a mapping from niceness to weight value (in a log-scale manner, just like the dB unit of loudness); CFS then weights time slices across n processes:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=1}^{n} \texttt{weight}_i} \cdot \texttt{sched\_latency} \text{ if } > \texttt{min\_granularity} \text{ else } \texttt{min\_granularity}$$

$$\texttt{vruntime}_k \mathrel{+}= \frac{\texttt{weight}_{\text{default}} = 1024}{\texttt{weight}_k} \cdot \texttt{physical\_runtime}_k$$

CFS uses a *red-black tree* to keep track of all ready processes for efficiency. When a process goes to sleep, it is removed from the tree. CFS also has many advanced features including heuristics and scheduling across groups and multiple CPUs.

Virtualizing the Memory: Memory Management

Apart from virtualizing the CPU, an OS must also be able to enable sharing of another important resource across processes - the memory. Processes require memory space to hold all of their runtime data, and it would be way too slow if we save/restore the entire memory content upon every context switch, especially when we are time-sharing the CPU (frequent switches). Hence, the OS must be able to do **space-sharing** of the memory: put all processes' in-memory data in the physical memory, and somehow let a process know how to *address* a byte it wants. Upon a context switch, we just switch the registers so that they point to correct addresses for the switched process. The OS must provide *protection*, *efficiency* in addressing, and a way to handle insufficient space if the physical memory is running out.

Abstraction of Address Space

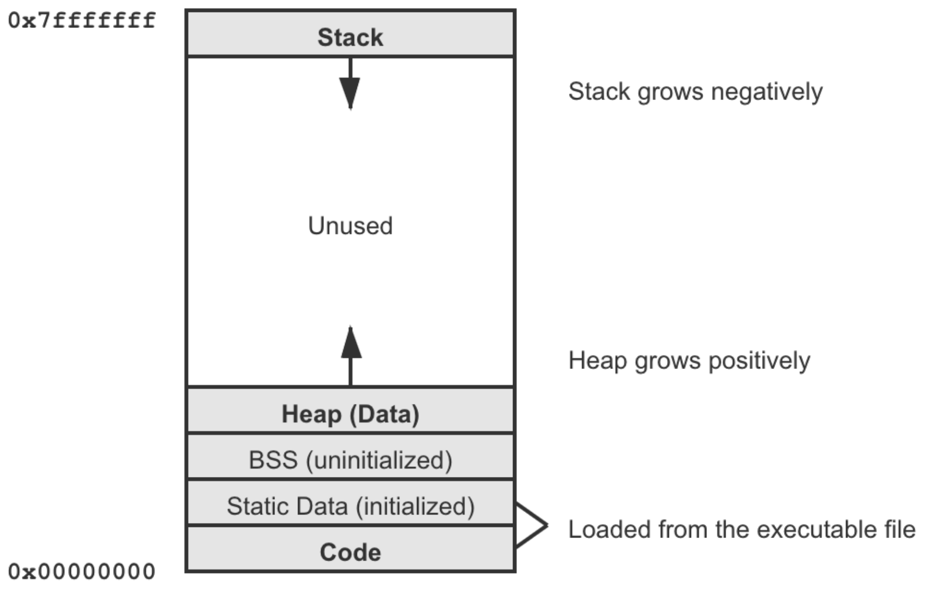
To enable the space-sharing of memory, we again turn to *virtualization*. We give each process an *illusion* of the ability to address across its own memory space, which could be huge and sparse. This abstraction is called the **address space** of the process - it is the running program's view of memory (**virtual memory**) and is completely decoupled with what the DRAM chip actually provides (**physical memory**). A program uses **virtual address** to locate any byte in its address space.

Address Space Layout

An OS must give a clear definition of the address space *layout* it expects. Conventionally, an address space contains the following *segments* and is structured in the following way:

- **Code**: the *program*'s machine code (the instruction); PC always points to somewhere in this segment
- **Static data**: holding global static data, could be further separated into initialized region (so read-only) and uninitialized region (so writes allowed)

- **Stack**: the *function call stack* of the program to keep track of where it is in the call chain; function calls form a stack of *frames*, where the local variables of a function resides in its frame; SP & FP always point to somewhere in this segment; stack grows negatively
- **Heap (data)**: for dynamically allocating runtime, user-managed memory (e.g., when calling `malloc()`); heap grows positively
- (optional) OS kernel mapped: the kernel could also be mapped into every process's address space (probably into the higher half), but this is an advanced topic



In reality, size of the virtual address space typically depends on the number of *bits* the hardware platform uses for addressing. If the hardware operates on 32-bit numbers, we could use an entire range from `0x00000000` to `0xffffffff`. The virtual address space could be very huge in size and very sparse (mostly empty) - this is not a problem. It is just the process's view on memory but we are not putting this space directly on physical memory at address 0. The OS will have some way to map the occupied bytes in the address spaces of all processes onto physical memory, which we will talk about in the next section.

This layout is just a convention for single-threaded processes. There could be other arrangements and things could be more complicated when a process becomes *multi-threaded*, which we will talk about in the concurrency section.

Memory APIs

These are the interfaces in C programming on UNIX systems for user programs to allocate memory. Other platforms might have different interfaces but they share the same spirit.

- *Implicit* (*automatic*) allocation of function call *stack* memory, containing local variables & local data
- *Explicit*, *dynamic* allocation of *heap* memory for large or long-lived, dynamic-size data
    - `brk()`, `sbrk()` - the primitive syscalls for changing (enlarging or shrinking) the heap data segment end-point bound; since memory allocation is tricky, user programs should never call these syscalls directly, but should rather use the library functions described below
    - `malloc()`, `calloc()`, `realloc()`, `free()` - normally, C programs link against a *standard library* which contains higher-level routines (wrapping over the `brk` syscalls) for allocating heap memory regions; these are not syscalls but are library functions with sophisticated allocation policies (using the syscalls as mechanism)
    - `mmap()`, `munmap()` - create/unmap a memory region for mapping a *file* or a *device* into the address space, often in a *lazy* manner, as an alternative I/O mechanism

With the `malloc()` + `free()` interface, here is a list of common memory errors in C programming:

- Forgetting to allocate memory and using an uninitialized pointer variable (*wild pointer*)
- Not allocating enough memory (*buffer overflow*)
- Not initializing allocated memory to known state (*uninitialized read*)
- Forgetting to free memory (*memory leak*)
- Accessing memory after it's freed (*dangling pointer*)
- Freeing memory repeatedly (*double free*)
- Freeing incorrect memory addresses

They often lead to the infamous *segmentation fault* (`SEGFAULT`) which happens when accessing a memory address the process shouldn't touch. Beware that code compiles & runs no segfault does not mean it runs semantically correctly - you could be accessing incorrect (but valid) memory addresses, which is still a serious bug.

Address Mapping & Translation

Every address a user program sees is virtual. The OS makes the virtual memory system *transparent* to users and implicitly *translates* any address a process asks for into physical memory. This is the procedure of **(hardware-based) address translation** and it implicitly implies the address mapping scheme the OS adopts. Here we list three address mapping and translation schemes. For efficiency, the OS makes use of hardware support (from a few register to full page table support) instead of simulating everything in software.

> The idea of **interposition** is really powerful and essential in systems. When virtualizing the CPU, the hardware timer sends timer interrupts to interpose the execution of a process to let the OS re-gain control periodically; this enables transparent time-sharing of the CPU. When virtualizing the memory, the hardware MMU interposes on each memory access to perform address translation; this enables transparent space-sharing of the memory.

We will discuss three address translation schemes, along with some complementory knowledge:

1. Base & Bound
2. Segmentation
3. Paging

Base & Bound

Let's first assume that the address spaces are small and compact, so it can fit in physical memory contiguously. With this assumption, the most intuitive way of address mapping is to simply **relocate** the address space to start at some offset in physical memory. The **base & bound** method (BB, or *dynamic relocation*, or *hardware-based relocation*) makes use of two hardware registers:

- *Base* register: the starting offset the OS decides to put the current process's address space at
- *Bound* register: the size (limit) of the current process's address space

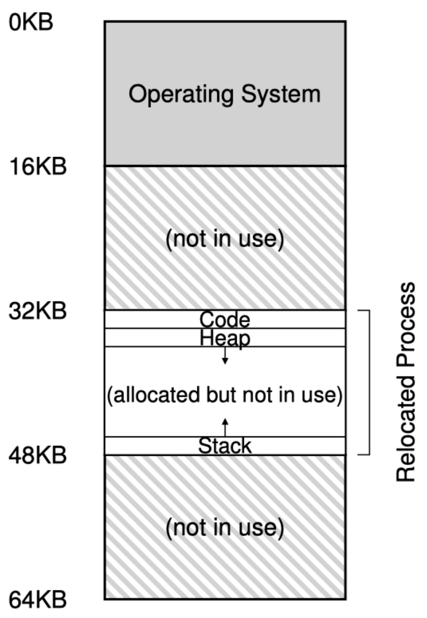


Figure 15.2: **Physical Memory with a Single Relocated Process**

Upon a context switch to a process, the two registers get loaded the base & bound values for this process. This step requires OS intervention to decide where to relocate (where is base). Along its execution, upon any memory access, the address translation is done by the hardware directly. This part of hardware (doing address translations) is often named the **memory management unit** (MMU).

The MMU algorithm for base & bound is obviously simple:

```
if virtual_addr >= bound:
    raise out-of-bound exception
else:
    physical_addr = virtual_addr + base
```

Without hardware support, purely software-based relocation is not so great. We could let a *loader* scan through the machine code of the executable file and change add a base value to every address it sees at process loading. However, this approach lacks protection and is not flexible.

Hardware/OS Responsibilities

A brief summary of hardware support we have talked about so far:

- Two privilege modes - OS code runs in privileged kernel mode and user program runs in user mode
- MMU registers, specifications, and the translation algorithm
- Privileged instructions only to be executed in kernel mode (e.g., updating base/bound)
- Privileged instructions to register trap/exception handlers
- Ability to raise hardware exception upon invalid memory access, ...

And the OS must take these responsibilities for virtual memory to work:

- Free space management: finding free space for new processes and reclaiming memory back at termination, via some sort of **free-list** data structures
- Upon context switch: setting proper MMU register states (e.g., base & bound values)
- Exception handling: decide what to do upon memory access exceptions the hardware reports, probably terminating the offending process

Segmentation

The base & bound approach has an obvious drawback: it requires that the physical memory has a big, contiguous chunk of empty space to hold the address space of a process. However, our address space could be huge and sparse, so it might not fit in as a whole. And all the unused bytes are still mapped, wasting a lot of memory space. **Segmentation** is a generalized form of base & bound which eases this problem.

A **segment** is a a contiguous portion of the address space. We could break the code, stack, & heap into three segments, then for each of them, apply base & bound independently. This way, only used memory is allocated space in physical memory.

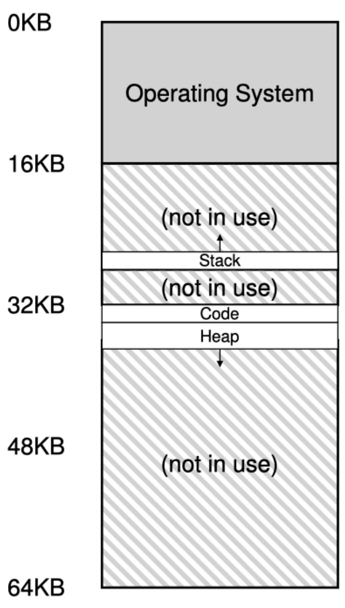


Figure 16.2: **Placing Segments In Physical Memory**

One problem thus arise: given a virtual address, how does the hardware know which segment it is referring to (so that it can do the base & bound check)? There are two approaches:

- *Explicit*: we chop the address space into fixed-sized segments, so that the highest several bits of a virtual address is the segment ID and the remaining bits is the offset in the segment (as we will see soon, this is an early form of paging)

  The MMU algorithm thus goes:

  ```
  segment_id = virtual_addr >> SEG_SHIFT
  offset = virtual_addr & OFFSET_MASK
  if offset >= bounds[segment_id]:
      raise out-of-bound exception
  else:
      physical_addr = offset + bases[segment_id]
  ```

- *Implicit*: the hardware detects how an address value is derived, and determines the segment accordingly (e.g., if an address value is computed by adding something to PC, it should be referring to the code segment)

Some extra support from hardware could be added:

- Negative-growth bit for stack: since stack grows negatively, this bit indicates offset should be calculated into a negative value
- Protection (permission mode) bits for sharing: some systems support sharing segments across processes (e.g.,

if code is the same, could share the code segment with `read-exec` permission)

Paging

TODO

Free-Space Management

In the above sections, we haven't really answered this question: how does the OS find proper free space when it needs to enlarge/allocate something? How to manage free space efficiently? We use the name **fragmentation** to describe the situation where there is space wasted in an address mapping scheme.

- *Internal* fragmentation: unused bytes in the address space getting mapped
- *External* fragmentation: small free spaces being left over between two mapped regions, which can hardly be used to map something else; there is no single contiguous free chunk, though the total number of free bytes is larger than the new segment

Different address translation schemes have different behaviors with regard to the two types of fragmentation. Fragmentation is hard to eliminate completely, but we can try to do better.

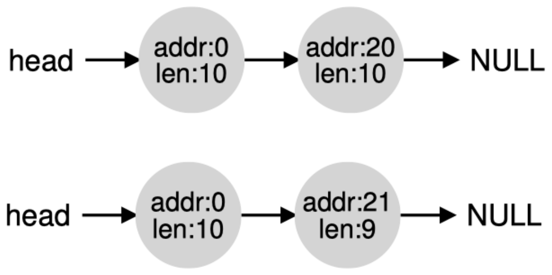
- Base & bound has both serious external fragmentation and internal fragmentation
- Segmentation has no internal fragmentation, but has external fragmentation (if w/o periodic *garbage collection*/*compaction*, useful but very expensive operations)
- Paging has no external fragmentation, but has a little bit of internal fragmentation (especially with large page size)

Note that the problem of **free-space management** is generic. It applies to how a `malloc()` library allocates new bytes on heap, how an OS allocates physical memory for new segments, etc. Here we list a few classic free-space management policies.

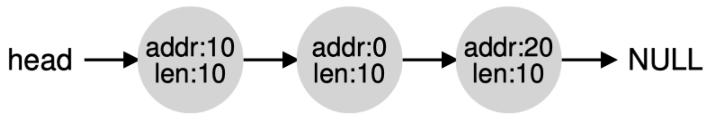
Splitting & Coalescing

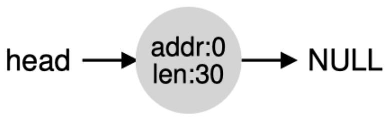
Suppose we keep a free list data structure as a linked list of free spaces. Two mechanisms play a significant role in all allocators:

- *Splitting*: when a small request comes, split an existing large chunk, return the requested size and keep the remaining chunk



- *Coalescing*: when a piece of memory is freed and there is free chunk next to it, *merge* them into one big free chunk





Basic Policies

These are the most basic, intuitive policies for memory allocation:

- **Best-Fit**: search through the free list and find the smallest free chunk that is big enough for a request (possibly with splitting)
  - ↑ Advantage: intuitive, nearly optimal
  - ↓ Disadvantage: expensive to do exhaustive search
- **Worst-Fit**: find the largest chunk, split and return the requested amount, keep the remaining chunk
  - ↑ Advantage: if using a max-heap, finding the largest chunk could be fast
  - ↓ Disadvantage: performs badly, leading to excess fragmentation
- **First-Fit**: find the first block that is big enough, split and return the request amount
  - ↑ Advantage: fast, stops early
  - ↓ Disadvantage: tends to pollute the beginning of the free list with small objects; *address-based ordering* could help
- **Next-Fit**: just like first-fit, but instead of always starting from the beginning, maintains a pointer to remember where it was looking last; start the search from there
  - ↑ Advantage: same efficiency advantage as first-fit
  - ↓ Disadvantage: requires keeping an extra state

> There is hardly any perfect free space allocation policy in general, because we could always construct a worst-case input of requests (*adversarial input*) to mess it up. We can only compare their pros and cons.

Segregated Lists

If a particular application has one (or a few) popular-sized request it makes, keep a separate list just to manage *objects* (i.e. free chunks of fixed size) of that size. All other requests are forwarded to a general memory allocator, say running one of the above policies. The **slab allocator** (see here) used in Solaris and Linux kernel is one such example.

- When the kernel boots up, allocate a number of *object caches* for possibly-popular kernel objects, e.g., locks, inodes, etc.
- When a given cache is running low on free space, itself requests some *slab*s of memory from a general allocator, the amount being a multiple of its object's size
- When a given cache is going empty, it releases some of the spaces back to the general allocator

Objects in the cache can also be *pre-initialized* so that the time for initialization and destruction at run time is saved.

Buddy Allocation

The **binary buddy allocator** (see here) is designed around splitting & coalescing in sizes of 2^i, allowing some internal fragmentation but making the coalescing procedure much more efficient.

- In the beginning, the allocator holds a one big space of size 2^N
- When a request comes, the search for free space recursively divides free space by 2, until the smallest power of 2 that is big enough for the request is found
- When that block is freed, the allocator checks if its *buddy* block (the one produced by the last split, its sibling block) is free; if so, it coalesces the two blocks; this procedure repeats recursively up the tree until the top of

the tree is reached, or until a level where the buddy is in use

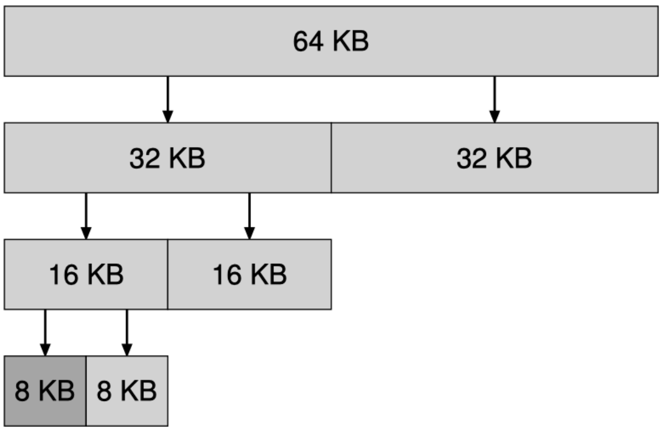


Figure 17.8: **Example Buddy-managed Heap**

We are particularly interested in *binary* buddy allocation, because it fits the binary address representation on most architectures so well. The address for a block differs with the address of its buddy block in just one bit (which bit depends on its level), so we can have highly-efficient implementations using bit-wise operations. Of course, there could be higher-order buddy allocation.

> For *scalability* issues on multiprocessors, *concurrent data structures* are being used in modern memory allocators to speed up concurrent accesses on the free list and to reduce the steps for *synchronization*. We will touch these in the concurrency section.

Advanced Paging

TODO

Advanced Page Tables

TODO

Translation Lookahead Buffer (TLB)

TODO

Caching and memory hierarchy?

Swapping

TODO

Concurrency: Multi-Tasking & Synchronization

TODO

Multi-CPU Scheduling

(Chapter 10)

TODO

Persistence: Storage Devices & File Systems

TODO

Advanced/Related Topics

TODO a list of notes, blogs, and books

Implementation (xv6, Phillip, Hux)

> I've made a brief OS history tree in XMind which is available here.