CS 452: Operating Systems
Slides

CS 452: Operating Systems

- Roster and passwords
- Our `pub` directory:

  `onyx:~jbuffenb/classes/452/pub`

  `pub/ch28/lock.c`

  Canvas

  GitHub
- Our lecture slides, table of contents, and code:

  `pub/slides/slides.pdf`

  `pub/slides/code.tar`
- Review syllabus:

  `http://csweb.boisestate.edu/~buff`

  `pub/syllabus/syllabus.pdf`
- Introduction (textbook)

Introduction to Operating Systems

- Our textbook's favorite word appears to be *virtual*: Not physically existing as such but made by software to appear to do so [Oxford University]. Its antonym is *real*.
- We also like *abstract*: A description of a concept that leaves out some information or details in order to simplify it in some useful way [The Free On-line Dictionary of Computing]. Its antonym is *concrete*.
- We use abstractions to create virtualizations.

A Lowest Abstraction Level: A CPU

- Wikipedia lists over 50 CPU companies. We'll assume a one-core chip, for now.
- What happens when you power-up a typical microprocessor?
  - Before the first machine-code instruction is executed.
  - PC/IP, MAR, MDR, IR, and the Fetch/Execute cycle.
  - BIOS code.
  - Bootloader code.
  - Kernel code.
  - Memory and devices.
  - Processes.
  - Logins and shells.

Another Lowest Abstraction Level: Main Memory

- Each (uniform) data-storage location has a (zero-origin) consecutive address.
- How many bits are in a word (location)? 8? 16? 32? 64? 60?
- How are bytes stored in a word?
- How many bits are in an address?
- We will assume byte-addressable memory, with 64-bit addresses.
- Sometimes, performance aspects, like word alignment and burst access, will be important.
- We will ignore details like dynamic refresh.

A Higher Abstraction Level: The Shell

- What happens when you run a program?
- The prompt.
- Programs.
- Processes.
- Libraries: static and shared.
- System calls: traps and privilege.
- Kernel entry points: devices and input/output (I/O).

Virtualization: CPU and Memory (1 of 2)

- Early OSs simplified I/O, by providing device abstractions (e.g., files, rather than disk tracks and sectors). OSs also simplified the job of loading a program into memory and executing it.
- However, when a program needed I/O, it would wait for the operation to complete. This wasted valuable resources: the CPU and memory.
- Rather than waiting for one program's I/O, the CPU should execute another program, at least for a while.
- This goal of *multiprogramming* was independent of whether a computer has multiple CPUs, cores, and/or users.

Virtualization: CPU and Memory (2 of 2)

- Each executing program should be presented with the illusion of being the *only* executing program, a simulation of exclusivity and isolation.
- Each program should execute on a *virtual* CPU with a *virtual* memory system, built atop the physical CPU and physical memory system. NB: Our current meaning of “virtual memory” encompasses much more than this simple abstraction.
- The technologies that enable this illusion are called *mechanisms*, while the various ways of managing them are called *policies*.

Concurrency

- Multiprogramming is a form of *concurrency*: doing multiple things at once. If executing programs must cooperate, total isolation is not possible: managed interaction is required.
- Concurrency can also be an illusion, built atop interleaved execution, independent of whether a computer has multiple CPUs, cores, and/or users.
- An OS employs various mechanisms and policies to provide concurrency and interaction: for one executing program, between executing programs, and within the OS itself.

Persistence

- When you power-up a computer, parts of the system retain their previous state: non-volatile memory (e.g., ROM and some types of RAM), disks, and other storage devices.
- Writable non-volatile devices are slow, compared to main memory. For better performance, an OS tries to batch/delay writes to such devices, in main memory, prior to “committing” them to the device.
- This delay poses a risk: a less than graceful power-down renders the device inconsistent with the rest of the system.
- There are clever ways to address this risk.

Design Goals

- Build abstractions, for simplicity.
- Minimize overhead, to maintain performance.
- Protect users from each other.
- Isolate users, by managing interaction.
- Provide a reliable computing environment.
- Manage energy consumption.
- Provide security against malicious attacks.
- Provide mobility and portability, easing adaptation to other hardware.

Some History

- Early Operating Systems: Just Libraries
- Beyond Libraries: Protection
  - system call
  - hardware privilege level
  - user mode
  - trap
  - trap handler
  - kernel mode
- The Era of Multiprogramming
  - minicomputer
  - multiprogramming
  - memory protection
  - concurrency
- The Modern Era: DOS, MacOS, Unix, and Linux

The Abstraction: The Process

- A *program* is typically a file on a disk. The content has a well-defined format (e.g., ELF), but is basically object code: machine instructions and data.
- A *process* is an executing program. The OS has loaded the program's content into virtual memory, and has created a virtual CPU to execute its machine instructions.
- The OS is *not* a process. Yes, the OS is machine instructions. Yes, the OS is in memory. Yes, the OS is executing. However, processes are part of the illusion created by the OS.

The Abstraction: A Process (1 of 2)

- The OS maintains information for each process: a process's *machine state*. This includes many items.
- Each process has its own virtual memory (aka, *address space*), containing its code, static data, stack, and heap. Part of the illusion is that each process's address space has the same range of virtual addresses, perhaps starting at zero.

The Abstraction: A Process (2 of 2)

- Each process has its own set of CPU registers, notably its:
  - PC: program counter (aka, IP: instruction pointer), the virtual address of the next machine instruction, in its address space, to execute
  - SP: stack pointer, the virtual address of the “top” of its call/return stack
  - FP: frame pointer, the virtual address of the other end of the top stack frame, for function-call arguments
- Each process has its own set of open-file descriptors, as well as other I/O information.

Process API

- An *Application Programming Interface* (API) is a set of function signatures provided by a software module.
- Here, we consider a stylized API, provided by an OS, allowing a process to manage other processes. We'll see a real one (e.g., `fork`, `exec`, and `wait`), in the next chapter:
  - Create: allocate and start a new subprocess
  - Destroy: stop and deallocate a subprocess
  - Wait: check or wait for a subprocess to stop
  - Control: do something to a subprocess (e.g., suspend it)
  - Status: get information about a subprocess

Process Creation: A Little More Detail

- A program (i.e., file content) is *loaded* from disk to virtual memory: machine instructions and static data. Depending on the OS, this may be *eager* (i.e., immediate and complete) or *lazy* (i.e., as needed).
- Other data areas, in virtual memory, are initialized (e.g., stack, heap, command-line arguments, and environment variables).
- Open-file descriptors are initialized (e.g., `stdin`, `stdout`, and `stderr`).

Process States

- Part of the machine state, maintained by the OS for each process, is the *process state*, a member of an enumeration:
  - Running: The process's instructions are executing on the real CPU. If it starts I/O it will become Blocked, else the OS can change it to Ready.
  - Ready: The process is neither Running nor Blocked.
  - Blocked: The process is waiting for some event (e.g., I/O to finish), at which time it will become Ready.

Data Structures

- The OS maintains data structures.
- We've seen that process information can be stored in a nested `struct`, a *Process Control Block* (PCB).
- PCBs can be stored in a *process list*.
- Part of a PCB is the saved register values of a non-Running process, its *register context*.
- When the OS changes which process is Running, register contexts must be saved and restored. This is called a *context switch*.

Interlude: Process API

- The primary Unix functions for managing processes are named `fork`, `exec`, and `wait`.
- Our textbook says they are "system calls," but they are really the names of families of functions, provided by `libc`, the C Standard Library. On Linux, it is the GNU C Library.
- These library functions make the actual system calls, and do other work.
- This extra level of indirection allows the syscall interface to evolve (e.g., on Linux, `fork` now makes the `clone` syscall).
- A famous aphorism of Butler Lampson goes: "All problems in computer science can be solved by another level of indirection" (the "fundamental theorem of software engineering"). An often cited corollary to this is, "...except for the problem of too many layers of indirection."

The `fork` System Call

- A *parent* process creates a *child* process (mostly a copy of itself), by calling `fork`.
- Both parent and child *continue executing* the same program.
- The call returns to the *same* virtual address in *both* processes!
- The two different return values allow both parent and child to deduce their identity: The parent's return value is the child's *process identifier* (PID). The child's return value is zero.
- Textbook code:

  `pub/ostep-code/cpu-api/p1.c`

The `wait` System Call

- A parent process can, and should, wait (i.e., block) for a child process to exit, by calling **`wait`** or **`waitpid`**.
- If the parent exits before a child, when the child exits it becomes a "zombie." Zombies are "adopted" by the **`init`** process (PID 1) for subsequent "reaping."
- Textbook code:

  `pub/ostep-code/cpu-api/p2.c`

Finally, The `exec` System Call

- Recall that after a `fork`, the child process continues executing the same program. However, you often want to execute a different program.
- An `exec` function call does just that. It replaces the current "process image" with a new one, and begins executing it.
- On Linux, there are six `exec` functions: `execl`, `execlp`, `execle`, `execv`, `execvp`, and `execvpe`:
  - `p`: a program name (without slashes) is searched for, via "path" (`$PATH`)
  - `l`: a "list" of command-line arguments are passed to `exec` as multiple arguments
  - `v`: a "vector" of command-line arguments is passed to `exec` as one argument
  - `e`: an "environment" of variable/value pairs is passed to `exec` as one argument
- Textbook code:
  `pub/ostep-code/cpu-api/p3.c`

Why? Motivating The API

- Unix and Linux separate `fork` and `exec` in an initially surprising way. One function *could* do both.
- However, separate functions allow a parent to have finer control over how its child executes a new program. Since a child starts as a copy of its parent, the parent can effectively set its child's:
  - working directory
  - command-line arguments
  - environment variables
  - file descriptors

  as if the parent ran it from a shell prompt.
- Indeed, separate `fork`/`exec` makes implementing a shell easy.
- Textbook code:

  `pub/ostep-code/cpu-api/p4.c`

Process Control And Users

- The last sections of this chapter vary between versions.
- Unix has other process-ish functions.
- A process can *signal* another process, with the morbidly named `kill` function, or shell builtin and program of the same name.
- A process can arrange to ''catch'' a signal, via the `signal` function, by executing a user-defined "handler" function upon arrival.
- An uncatchable signal, named `SIGKILL`, really does kill (terminate) a process.
- Other signals interrupt (`SIGINT`) or stop/pause/suspend (`SIGTSTP`) a process.
- A stopped process be resumed or backgrounded, with the shell's `fg` or `bg` builtin commands.

Useful Tools

- All of these programs and functions are described in `man` pages. Read them.
- Programs `top` and `ps` show information about processes.
- Programs `kill` and `killall` send a signal to processes.
- Every process has a *user identifier* (UID), upon which protection, isolation, and security are based.
- PIDs, UIDs, and signals are integers.
- UID 0, often called “root” or “superuser,” is for system administration. It enjoys unrestricted access.
- Programs `su` and `sudo` execute a command as a different user, often UID 0.

Mechanism: Limited Direct Execution (1 of 2)

- Recall that we plan to accomplish our illusion by executing programs on virtual CPUs, each with a virtual memory.
- Here, we describe a mechanism to execute multiple virtual CPUs atop the physical CPU.
- We will time-division multiplex, or *time share*, the virtuals on the physical.
- We could simulate CPUs, with software that interprets machine instructions, but that would be too slow. We want a virtual CPU to execute its program at full (i.e., hardware) speed.

Mechanism: Limited Direct Execution (2 of 2)

- Question: How do we get the executing program to relinquish control back to the multiplexer (i.e., OS), so another program can be executed?
- Answer:
  - Require the program to call OS code, occasionally (e.g., for I/O).
  - Preempt the program, periodically, with a hardware-timer interrupt.

Basic Technique: Limited Direct Execution

- The “Direct Execution” part means: Jump to, or call, the program’s code, and let it run. Simply load the physical CPU’s PC register with the virtual address of a machine instruction of the program.
- The “Limited” part means:
  - Restrict what the program can do, so it behaves: Prohibit its execution of certain machine instructions.
  - Restrict how long the program runs, so it shares: Interrupt its execution with a hardware timer.

Problem #1: Restricted Operations (1 of 3)

- To control behavior and sharing, code running on a virtual CPU must be constrained:
  - No subverting the illusion (e.g., changing the preemption timer).
  - No accessing hardware directly.
  - No circumventing user-based OS protections.
- This requires hardware:
  - Partition the machine-instruction set into two (or more) privilege-level sets (e.g., *kernel* and *user*).
  - Augment the CPU's register set with enough extra bits to select a privilege level. This can simply be a bit in an existing "status" register, selecting *kernel mode* or *user mode*.

Problem #1: Restricted Operations (2 of 3)

- Upon power-up, the CPU is put into kernel mode.
- In kernel mode, instructions from any privilege-level set can be executed.
- As part of the context switch to a virtual CPU's code, execute an instruction to put the CPU into user mode.
- In user mode, only instructions from the user privilege-level set can be executed, without causing a *trap* (aka, software interrupt).

Problem #1: Restricted Operations (3 of 3)

- A trap is like a function call, but jumps only to a *trap handler* within the kernel, via an indexed *trap table* (aka, vector).
- A trap puts the CPU into kernel mode.
- A trap can also be caused by a special machine instruction (e.g., `int` or `syscall`), making a system call. An operand might be an index to the trap table.
- A trap handler analyzes the trap's cause. A permissable syscall is performed, as requested, then the trap handler executes a "return from trap" instruction (e.g., `iret`). Otherwise, the virtual CPU (i.e., process) is terminated.
- An `iret` instruction puts the CPU back into user mode.

Problem #2: Switching Between Processes (1 of 3)

- The first part of this problem is to switch from the running process back to the OS:
  - Trust the process to voluntarily make a special "yield" syscall, occasionally.
  - Trust the process to make ordinary syscalls (e.g., for I/O), occasionally, as part of its normal work.
  - Assuming the process cannot be trusted, or is malicious, and will not relinquish the CPU via a syscall, establish a hardware-timer interrupt, to periodically interrupt the executing process. The resulting hardware trap is much like a syscall.

Problem #2: Switching Between Processes (2 of 3)

- Regardless of whether the trap is from software or hardware, the OS begins by saving enough of the process's state (e.g., the content of a few important registers), so it can be restored prior to returning.
- If the trap is an ordinary syscall, the OS initiates that action (e.g., I/O).

Problem #2: Switching Between Processes (3 of 3)

- The second part of this problem is for the OS to decide which process should be resumed, according to a scheduling policy.
- If it's not the same process that was executing, a context switch to a different virtual CPU is performed.
- NB: The trap/syscall save/restore actions are separate from, and simpler and faster than, those of a full context switch.
- Finally, the OS can set the hardware timer, restore previously saved registers, and reenable interrupts with an `iret`.

Worried About Concurrency?

- What if an interrupt occurs while a previous trap or a syscall is happening?
- When a hardware or software trap happens, the CPU automatically disables interrupts. This gives the OS time to prepare for another, before enabling them.
- Interrupts must not be disabled for long, or important events (e.g., I/O completion) could be missed.
- A syscall or context switch might happen in about a microsecond.
- We will study these concurrency challenges, in later chapters.

Scheduling: Introduction

- We now know how the OS kernel can:
  - regain control of the physical CPU from the process executing on a virtual CPU, via hardware and software traps
  - change which virtual CPU is to execute on the physical CPU, via context switches
- Now we need to learn how the OS can choose a process to execute next, via a scheduling policy (aka, algorithm).
- A branch of applied mathematics, called *operations research*, started in the 17th century, provides our foundation. Consider the problem of optimizing an assembly line's productivity.

Workload Assumptions

- The processes currently executing on virtual CPUs are called the *workload*.
- Our textbook uses the word *job* synonymously with the word "process." Unix does not!
- For now, we make (somewhat unrealistic) assumptions about a workload's jobs:
  - They run for the same amount of time.
  - They arrive at the same time.
  - Once started, each runs to completion.
  - They use only the CPU (i.e., no I/O).
  - The time to run each is known.

Scheduling Metrics

- Before considering and comparing scheduling policies, we need quantitative metrics (i.e., a numeric measurable result).
- The *turnaround time* of a job is the time at which the job completes minus the time at which the job arrived in the system.
- This is a performance metric. There are other kinds.

First In, First Out (FIFO) (1 of 2)

- This is synonymous with First Come, First Served (FCFS).
- Since our assumed workload's jobs arrived at the same time, each has $T_{arr} = 0$, so they can be run in any order.
- Suppose we have three jobs: A, B, and C; we run them in that order; and each runs for 10 seconds.
- The completion times are 10, 20, and 30.
- The average turnaround time, in seconds, is simply:

$$\frac{10 + 20 + 30}{3} = 20$$

First In, First Out (FIFO) (2 of 2)

- Now, relaxing one assumption, suppose the run times are 100, 10, and 10.
- Run in the same order, the completion times are 100, 110, and 120.
- The average turnaround time is:

$$\frac{100 + 110 + 120}{3} = 110$$

- How might we improve this?

Shortest Job First (SJF) (1 of 2)

- Suppose we the run previous workload in other orders.
- 10, 100, then 10:

$$\frac{10+110+120}{3}=80$$

  Better!
- 10, 10, then 100:

$$\frac{10+20+120}{3}=50$$

  Best!
- In fact, we could *prove* that running jobs in order of increasing run time is optimal, for average turnaround time.

Shortest Job First (SJF) (2 of 2)

- Suppose we also relax our assumption about arrival time:

job	T_{arr}	T_{run}
A	0	100
B	10	10
C	10	10

- Since A arrives first, it is the shortest job at that time, so it is run first, and the average turnaround time is:

$$\frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33$$

- We would have been better off if A arrived 20 seconds later!

Shortest Time-to-Completion First (STCF)

- This is synonymous with Preemptive Shortest Job First (PSJF).
- We can avoid the previous anomaly, by relaxing another assumption: Once started, a job runs to completion.
- In the previous example, this would allow the OS to preempt A, when B and C arrive at time 10. A would not get to continue until B and C were done. The resulting (optimal) average turnaround time would be:

$$\frac{(120-0)+(20-10)+(30-10)}{3} = 50$$

A New Metric: Response Time

- STCF is fine if jobs do not perform I/O, but this assumption is especially unrealistic in interactive environments.
- For example, when you press a key, you expect to see the screen change immediately.
- The *response time* metric tries to capture this idea: the difference between a job's arrival time and the time at which it begins execution.
- This definition is suspicious: the job might not "respond" immediately, and it might be preempted.
- For the previous example, the average response time, in seconds, is:

$$\frac{(0-0)+(10-10)+(20-10)}{3} = 3.33$$

- The next scheduling policy tries to improve response time.

Round Robin (RR)
(1 of 3)

- Rather than only preempting a running job when a “shorter” job arrives, we can preempt periodically to prevent *any* job from waiting too long.
- This scheme can provide *fairness*, and prevent *starvation*.
- The word “periodically” means each job gets a *time slice* (aka, *scheduling quantum*), a multiple of the timer-interrupt period (e.g., 100ms).

Round Robin (RR) (2 of 3)

- Suppose we have:

job	T_{arr}	T_{run}
A	0	5
B	0	5
C	0	5

- Shortest Job First would have a average response time of:

$$\frac{0+5+10}{3} = 5$$

- Round Robin, with a 1-second time slice, would have a average response time of:

$$\frac{0+1+2}{3} = 1$$

Round Robin (RR) (3 of 3)

- Average response time can be reduced by reducing the length of a time slice, but only so far. We have been ignoring the time required for a context switch. Eventually, it would dominate.
- A context switch has to not only save and restore process states, but it also must flush the CPU's caches and virtual-memory buffers.
- What about average turnaround time? In the example above:

$$\frac{(5+4+4)+(5+5+4)+(5+5+5)}{3}=14$$

  Ugh! The price of fairness.

Incorporating I/O

- Our I/O assumption is silly. With no input, output is always the same. With no output, you can't tell it ran.
- When the running job begins I/O, it becomes *Blocked* on I/O, awaiting completion. Since the physical CPU would be idle, the scheduler does a context switch, to a different job, as if a timer-interrupt occurred,
- When that I/O ends, the *Blocked* job becomes *Ready*, and may even get to run immediately.
- For scheduling policies that choose a job based on some flavor of run time (e.g., SJF), a job can be split into sub-jobs, based on I/O activity.
- This *overlap* improves global performance.

No More Oracle

- Our run-time assumption is also silly. The Halting Problem is undecidable, and pre-computing a program's run time is even harder.
- We could try to use heuristics, like the program's previous run time. We won't even bother.

Scheduling: The Multi-Level Feedback Queue (MLFQ)

- This policy tries to optimize both turnaround time and response time, without making silly assumptions.
- It is based on a general, but common, heuristic: use past behavior to predict the future.

MLFQ: Basic Rules

- This policy uses a set of queues, each having a priority value (i.e., an integer). A higher value means "higher priority."
- To choose a job to run, the OS uses Round Robin on jobs in the highest-priority queue.
- Also, the OS can move a job from one queue to another, partially based on a job's past behavior (e.g., I/O activity).

Attempt #1: How To Change Priority

- This is a first stab at an adjustment algorithm.
- When a job arrives, give it the highest priority.
- If a job runs for its entire time slice, reduce its priority.
- If a job preempts itself (e.g., with I/O), don't change its priority.
- Sadly, this algorithm overly favors interactive jobs, starving batch jobs.
- Also, a job might perform unneeded I/O, just to get a higher priority.
- Also, an initially batch job that becomes interactive is not given a priority increase.

Attempt #2: The Priority Boost

- This refines Attempt #1, by adding a rule.
- After some time period, reset all jobs to the highest priority.
- This change avoids starvation, due to Round Robin. It also allows a low-priority job to become and remain a high-priority job, if it becomes I/O active.

Attempt #3: Better Accounting

- This attempt changes Attempt #1.
- It prevents fake I/O from maintaining a job's priority.
- Each job is assigned a maximum time that it can spend in each queue (i.e., at each priority).
- After a job is run, regardless of whether it preempted itself, if it has exhausted its time at its current priority, lower its priority, and repeat.
- This ignores I/O altogether, and avoids starvation due to priority boosting and Round Robin.

Tuning MLFQ and Other Issues

- Real schedulers are configurable, because there are no best values.
- How many queues? Solaris defaults to 60.
- What time limit for each queue? Perhaps, 10-20ms for high priority and a few hundred milliseconds for low priority.
- How often should priority be reset? Perhaps, once a second.

Scheduling: Proportional Share

- This is also called Fair-Share Scheduling.
- Rather than focusing on turnaround time or response time, this policy tries to give each job a certain percentage of physical CPU time.
- The idea is to give each job a certain number of *tickets*, according to its assigned CPU share. A randomized *lottery* draws a ticket, choosing a job.

Basic Concept: Tickets Represent Your Share

- A ticket can simply be an integer. A job might be given a range of them, for a percentage of a CPU.
- A lottery can simply be the generation of a pseudorandom integer.
- The quality of randomness, and the number of lottery draws, promises (but does not guarantee) probabilistic correctness.

Ticket Mechanisms

- Tickets can be the basis for a quota system, or treated as currency (i.e., money).
- A user can allot its tickets to its jobs.
- A job can allot its tickets to sub-jobs.
- A client job can allot some of its tickets to a server job, as part of a request.

Implementation, Example, and Evaluation

- The algorithms and data structures are simple. Binary search or hashing can increase performance.
- Imagine we have two jobs, with the same arrival and run times, which are given the same number of tickets. We can define a simple *fairness* metric to be the quotient of the first and second completion times. A value of 1.0 means perfect fairness.
- In practice, how many tickets should be allocated to each job? It's an open question.

Stride Scheduling

- This is a deterministic variation of lottery scheduling: no randomness.
- Rather than a number of tickets, each job has a *stride*: some big integer divided by the job's number of tickets. Stride is basically the reciprocal of ticket count: A high-priority job has a small stride.
- Each job also has a counter, called its *pass*. When a job runs, its pass is incremented by its stride.
- To choose a job to run, the scheduler simply chooses the one with the lowest pass.
- There's a catch. A newly arrived job must be assigned a pass, and it shouldn't be zero. Perhaps, it should be the lowest active pass (i.e., the pass of the last job run). Would this be fair?

The Linux Completely Fair Scheduler (CFS) (1 of 7)

- The Linux CFS achieves fairness in a different way than we've seen in previous proportional-share policies.
- It also is designed to be efficient and scalable. In a Google datacenter study, its overhead was measured to be about 5% of CPU time.
- Each process has a *virtual runtime* (`vruntime`), similar to Stride Scheduling's pass, but it is usually proportional to the process's real run time.
- To choose a job to run, the scheduler simply chooses the one with the lowest `vruntime`.

The Linux Completely Fair Scheduler (2 of 7)

- The running job's time slice is computed from the parameter `sched_latency` (e.g., 48ms) divided by the number of runnable jobs, but not below `min_granularity` (e.g., 6ms). This threshold limits overhead.
- Since timer-interrupt frequency is constant (e.g., every 1ms), the time slice is approximate.

The Linux Completely Fair Scheduler (3 of 7)

- We can see this in recent (linux-5.11.22) kernel code:

  include/linux/sched.h

```
struct sched_entity {
    ...
    u64 vruntime;
    ...
}
...
struct task_struct {
    ...
    struct sched_entity se;
    ...
}
```

  kernel/sched/fair.c

```
unsigned int sysctl_sched_latency
  = 6000000ULL; //  6.00ms
unsigned int sysctl_sched_min_granularity
  =  750000ULL; //  0.75ms
```

The Linux Completely Fair Scheduler (4 of 7)

- Here's where a child process "inherits" its parent's `vruntime`:

  kernel/sched/fair.c

```
// called on fork with the child task as
// argument from the parent's context
//  - child not yet on the tasklist
//  - preemption disabled
static void task_fork_fair(
        struct task_struct *p) {
    ...
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se, *curr;
    ...
    cfs_rq = task_cfs_rq(current);
    curr = cfs_rq->curr;
    if (curr) {
        update_curr(cfs_rq);
        se->vruntime = curr->vruntime;
    }
    place_entity(cfs_rq, se, 1);
    ...
}
```

The Linux Completely Fair Scheduler (5 of 7)

- Unix has a command to adjust job priority: `nice`. Its priority argument is an integer between −20 (high) and +19 (low). The default is 0.
- The argument is an index to the array `sched_prio_to_weight[40]`, in `kernel/sched/core.c`, as shown in our textbook.
- The indexed value is used to adjust a job's time slice, and the growth of its `vruntime`, as shown in our textbook. Its priority increases because its `vruntime` increases in smaller increments than that of lower-priority processes.

The Linux Completely Fair Scheduler (6 of 7)

- Any scheduler employs priority queues, as part of whatever policy it uses. For a small number of jobs (e.g., less than 100), a linear linked-list implementation is fine. However, the Linux scheduler might need to juggle thousands of jobs, so a fancier implementation makes sense.
- The CFS uses red-black trees, a clever form of height-balanced tree, for logarithmic performance. So does the GNU C++ library.

The Linux Completely Fair Scheduler (7 of 7)

- Should the time a job spends blocked on I/O, or otherwise sleeping, be counted as run time?
- If not, when the job becomes *Ready*, its low `vruntime` would make it a high priority job, and another job that arrived at the same time as the first would starve.
- So, the CFS increases the `vruntime` of such a job to the lowest `vruntime` of jobs in the queue.

The Abstraction: Address Spaces (1 of 2)

- We've learned how to virtualize and multiplex a physical CPU, with the help of privileged machine instructions and hardware-timer interrupts.
- Each virtual CPU (i.e., process) now needs its own virtual memory, an abstraction built upon physical memory.

The Abstraction: Address Spaces (2 of 2)

- Every address visible to a process will be a virtual address. A process will not be allowed to access physical memory with a physical address.
- Clearly, this requires address translation. It must be enforced, transparent, small, and fast. Easy, right?
- This is all part of the illusion cast by an OS kernel. Each process sees a huge, contiguous, uniform, isolated, and protected address space.

Early Systems (1 of 2)

- Originally, the OS (kernel) would boot into low memory, and run a program by loading it into higher memory and jumping to it.
- There was no memory protection or address translation.
- A program could call a kernel function as if it was a library function.
- If a program corrupted the kernel, the computer would be rebooted.

Early Systems (2 of 2)

- A little later, this scheme was slightly extended to support execution interleaving of multiple programs.
- At first, a context switch would completely:
  - save the currently executing program's memory image to disk
  - restore the image of the next program to execute

  This was too slow.
- Later, with larger memories, program images could stay in memory, and a context switch would just affect registers. This was much quicker.
- There was still no memory protection or address translation.
- Also, each of a program's addresses needed adjustment, based upon its load address. This is called *relocation*.

The Address Space (1 of 2)

- Recall that each process (i.e., virtual CPU) has memory segments for its code, statically allocated data, heap, and stack. A process with more than one thread has that many stacks.
- If a code segment always starts at the same virtual address (e.g., `0x0`$\cdots$`00`), we can avoid load-time relocation. The heap and stack can go anywhere, but typically grow toward one another: heap just above static data, stack at the “top” of memory.

  `pub/ch13/mem-addr.pdf`

The Address Space (2 of 3)

- What is the top of memory? What is its address?
- The highest virtual address can be *much* larger than the highest physical address of physical memory.
- This is because a virtual address only needs to translate to a physical address *when the virtual CPU accesses it*.
- For example, a process's illusion might allow 63-bit virtual addresses (8 exabytes or 8 billion gigabytes), which are mapped to (aka, backed by) a computer's 4 gigabytes of physical memory. 4GB requires 32-bit addresses.
- Note that a 64-bit processor (e.g., x86-64 and ARMv8) might *really* allow only 48-bit addresses, for "only" 256 terabytes.

The Address Space (3 of 3)

- Indeed, this is today's x86-64 Linux virtual-memory layout, for a process's user region:

  `pub/ch13/mem-user.pdf`

  It has 47-bit virtual addresses, starting at 0. Notice that there is much "empty" space. Try: `pmap $$`.
- The kernel has the rest of the 64-bit virtual addresses, with bit 47 set.
- Note that current CPUs can only generate 48-bit physical addresses.

  `pub/ch13/mem-proc.pdf`

Goals

- Earlier, we used the word "transparent." Sadly, many people use it to mean its opposite, "visible." We want our illusion to be transparent: a process is unaware of (i.e., cannot see) virtual-to-physical address translation.
- Such translation should be efficient, in time and space. We will use a hardware cache to make a virtual-memory access as fast as a physical-memory access, in most cases. In cache-miss cases, our data structures will be compact.
- Complete transparency will give us process protection and isolation, for "free." A wayward or malevolent process cannot subvert an invisible mechanism.

Interlude: Memory API

- This is mostly a review of material from CS 253.
- Memory can be allocated from different areas in a process's virtual address space:

  `pub/ch13/mem-addr.pdf`

  `pub/ch14/Mem/Mem.c`

Mechanism: Address Translation (1 of 3)

- Address-translation hardware has evolved, starting in the 1950s. This chapter describes the earliest scheme, called *base and bound*.
- This scheme augments the CPU with a *memory-management unit* (MMU), containing two registers: a *base register* and a *bound register*:

  `pub/ch15/BaseBound.pdf`
- The MMU translates a virtual address, from the ALU/register bus, into a physical address, which is latched by the MAR.

Mechanism: Address Translation (2 of 3)

- During a context switch, kernel-mode instructions save/restore a process's lowest physical and highest virtual addresses from/to the base and bound registers, (resp.).
- Every virtual-address access is compared to the bound and added to the base, to validate and translate it.
- An out-of-bounds address aborts an access. This allows the kernel to terminate the offending process. This is a simple form of what happens when your program gets a "seg fault."

Mechanism: Address Translation (3 of 3)

- This scheme implements *dynamic relocation*, which is similar to, but a replacement for, the static relocation we discussed earlier.
- This allows the kernel to relocate a process's memory, during a context switch. Process memory can be rearranged to reduce external fragmentation.
- The kernel can also *swap-out* a process, to disk, during a context switch. Subsequently, the kernel can *swap-in* the process to a different region of physical memory.

Segmentation (1 of 2)

- With base and bound, both of a process's virtual and physical address spaces are contiguous (i.e., no gaps).
- The next advance in the evolution of virtual-to-physical address translation, in the 1960s, allows a process's physical address space(s) to be disjoint. This reduces, and helps repair, fragmentation.
- Also, an "empty" space in a process's virtual memory need not be mapped to physical memory. This greatly conserves memory.

Segmentation (2 of 2)

- Boarding the bandwagon of incremental innovation: If one base/bound register pair is good, four must be better! We'll use one pair for each memory segment: code, static data, stack, etc.

  `pub/ch16/Segmentation.pdf`
- A virtual address is translated according to exactly one base/bound register pair. There are two approaches for selection.
- The high two bits of a virtual address could explicitly select one of the pairs.
- The way in which a virtual address is formed could implicitly select one of the pairs. For example, if a `push` instruction gates the content of the stack-pointer register into the MMU, the control unit would select the pair for the stack segment.

Support for Sharing

- Refinements to segmentation added metadata to each segment, implemented as bits in additional registers.
- A segment could be marked as “grow upward” or “grow downward,” to support stack-segment resizing.
- A segment could be marked as: “read/write,” “read-only,” or “executable.” This enabled sharing of a code segment among multiple processes; a huge savings.
- A “sticky bit” could mark a segment as “executable” and “keep in memory,” to prevent the repeated reloading of popular programs or libraries. It was set via a file’s metadata of the same name. The term has a different meaning, now.

Fine-grained vs. Coarse-grained Segmentation (1 of 2)

- Further features foretold future fine-tuning.
- If four segments were good, more would be better!
- The Multics OS and the Burroughs B5000 machine supported thousands of segments, but with a twist.
- Segment bases, bounds, and metadata could be stored in memory, in addition to the CPU's MMU registers.

Fine-grained vs. Coarse-grained Segmentation (2 of 2)

- During a machine-instruction's execution, which required virtual-address translation, a segment's information might need to be transferred from memory to the MMU.
- Uh oh! A memory access might require multiple memory accesses.
- Let's push this problem onto our stack. Instead, in the next chapter, let's consider how a kernel can manage allocation, deallocation, and fragmentation of these segments.

Free-Space Management (1 of 2)

- Managing a set of uniform resources is easy. It's difficult when they differ.
- For example, managing same-size blocks of memory is easy (e.g., as with Lisp's `cons`). When a client can request a particular size (e.g., as with `malloc`), internal and external fragmentation must be mitigated.
- *Internal fragmentation* occurs when a larger-than-requested block is allocated, to simplify deallocation. The extra space is wasted.
- *External fragmentation* occurs when a sequence of allocations and deallocations result in a patchwork of free space. A subsequent request may only be satisfied by a contiguous fraction of the remaining free space.

Free-Space Management (2 of 2)

- Many smart people have given this problem much thought. We'll make some simplifying assumptions, for our domain.
- We assume explicit allocation and deallocation (i.e., no garbage collection).
- We focus on external fragmentation, caused by allocations of varying size.
- We assume an allocated block cannot be "moved" (i.e., no compaction). This allows pointers to data within a block.
- We assume an allocated block *contains* no metadata. Such data might be stored elsewhere. Real allocators, especially those providing debug support, violate this assumption.
- We assume the initial *pool* of free memory is contiguous (aka, *heap* or *arena*).

Low-level Mechanisms

- Since a client manages allocated blocks, we need only manage free blocks. We could use a linked-list of pointers to free blocks, but that is wasteful (and somewhat circular).
- Since free blocks are unused, we can link them together by embedding pointers *within* the free blocks.
- Our textbook has numerous detailed pictures of this, showing how *splitting* and *coalescing* occur. Splitting might be part of allocation. Coalescing might be part of allocation or deallocation.
- Now, we need a way to "match" an allocation request of a particular size with a free block, perhaps after splitting and/or coalescing.

Basic Strategies

- Imagine that the allocator has a list of free blocks of sizes 6, 14, 19, 11, and 13 (KB), in that order, and a request of size 12 is made.
- For *best fit*, the allocator would choose the block of size 13, leaving a free block of size 1. This is probably external fragmentation.
- For *worst fit*, the allocator would choose the block of size 19, leaving a free block of size 7. This probably is not external fragmentation.
- For *first fit*, the allocator would choose the block of size 14, leaving a free block of size 2. This tends to fragment the front of the list,
- For *next fit*, the allocator records the position of the last allocation. Suppose it was between the size 11 and 13 blocks. The allocator would choose the block of size 13.

Other Approaches

- Rather than just one free list, you can maintain *segregated lists* of different sized blocks, to reduce the time needed to search a list. Indeed, you might be able to deduce, and advantage, a pattern of allocation-size requests.
- What about other data structures?
- The *Buddy System* is a more complex strategy, explored further in a homework assignment.
- A Buddy allocator begins with a single large free block (e.g., 64KB). To satisfy a request, a free block is repeatedly split, into buddies, until a block of proper size results. For example, a request of 12KB would leave the following free blocks: 32KB and 16KB, resulting in 4KB of internal fragmentation. Upon deallocation, free buddies are coalesced.

Paging: Introduction

- In this chapter, we resume our quest for fine-grained virtual-address translation, without too much memory-access overhead.
- Our first improvement, over segmentation, is to adopt the allocation protocol of Lisp, the second programming language, from 1958: fixed-sized allocations.
- Our *pages* are all the same size. Goodbye external fragmentation!

A Simple Example And Overview (1 of 2)

- A process's virtual memory is partitioned into fixed-size *virtual pages*, each with a *virtual-page number* (VPN).
- These are mapped to *physical pages* (aka, *frames*), of the same size, each with a *physical-page number* (PPN) (aka, *frame number* (PFN)).
- Based on the page size, some number of high-order bits of a virtual address are the VPN, which indexes a per-process *page table*, to get a PFN.
- The remaining, low-order, bits (i.e., the *offset*) are combined with the PFN to form the physical address to access.
- The needed and/shift/index/shift/or operations are simple and fast.

The Example (1 of 5)

- Suppose we have 48-bit virtual addresses and a (typical) page size of 4KB. Since the offset needs 12 bits, 36 bits are available for page numbers. A page table, for each process, containing 2^{36} (perhaps 16-byte) entries is infeasible. We need a plan.
- Eventually, our plan will be to represent each page table as a sparse tree (i.e., a table of pointers to tables). Yes, that means even more memory accesses!

The Example (2 of 5)

- For now, imagine a loop that zeroes an array's elements. We focus on memory accesses that occur while executing just one machine instruction in the loop's body:

  `pub/ch18/a.c`

  `pub/ch18/a.s`
- Notice that `a` and `i` are statically allocated. If they were automatically allocated (e.g., local variables, on the stack), their stack-frame addresses would have to be calculated.

The Example (3 of 5)

1. The virtual address in the program counter (`0x40111e`) is AND-ed and shifted to obtain the VPN of the page containing the next instruction, and AND-ed to obtain its offset.
2. The physical address in the MMU's page-table base register is indexed by the shifted VPN to form the physical address of the PTE.
3. The physical address in the PTE is read from memory, shifted, and OR-ed with the offset, to form the physical address of the next instruction.
4. The next instruction is read from memory, and the program counter is incremented (eventually) by eleven.

The Example (4 of 5)

5. The `movl` instruction has operands:
   - the 32-bit virtual address of the start of the array (`0x00404060`)
   - an index register (`%eax`), already containing the value of `i`
   - the base-type size of the array (4)
   - the 32-bit value to store in the array (0)
6. The virtual address of the 4-byte destination of the `movl` is computed by adding the address of the start of the array to the product of the index value and base-type size. It is then AND-ed and shifted to obtain the VPN of the page containing the destination, and AND-ed to obtain its offset.

The Example (5 of 5)

7. The physical address in the MMU's page-table base register is indexed by the shifted VPN to form the physical address of the PTE.
8. The physical address in the PTE is read from memory, shifted, and OR-ed with the offset, to form the physical address of the destination.
9. Finally, a 4-byte 0 is written to the destination's physical address in memory.

Paging: Faster Translations (TLBs)

- With segmentation, the several base/bound register pairs could be within the CPU/MMU (maybe a coprocessor).
- With paging, the number of virtual pages is so large, the virtual-to-physical mapping pairs won't fit in the MMU. They are stored in memory. The MMU's *page-table base register* (PTBR) contains the physical address of this data structure.
- If it is a *linear* page table, it is huge, and every virtual-memory access requires two physical-memory accesses (yuck!).
- If it is a *multi-level* page table, it is much smaller, but every virtual-memory access requires at least two physical-memory accesses (double yuck!).
- The solution: add more MMU hardware.

TLB Basic Algorithm (1 of 3)

- In addition to MMU's mundane and/or/shift/index logic, we add a cache.
- The cache records results of previous virtual-to-physical address translations. For historical purposes, it is called a *Translation-Lookaside Buffer* (TLB).
- A TLB only needs to map a VPN to a PPN/PFN. The offset can be AND-extracted before, and OR-injected after, each translation. This observation reduces the "width" of a TLB.

TLB Basic Algorithm (2 of 3)

- A TLB is a sort-of memory, but it is too small to hold every translation. It can't just use a VPN as an address/index. So, an *associative memory* (aka, *content-addressable memory*) is used.
- In Java-API terms, an associative memory can be thought of as a table, implementing the `Map<VPN,PFN>` interface.
- In hardware terms, an associative memory isn't addressable. It's just a set of storage cells, each containing a key/value pair (i.e., `<VPN,PFN>`).
  To read a key's value, that key is hardware-compared to the key part of every cell's content. A matching cell's value part is the result of a "hit."
  No match means "miss."

TLB Basic Algorithm (3 of 3)

- Our textbook's Figure 19.1 is pseudocode for a TLB-aided translation. The basic idea, of course, is to check the cache before the page table.
- A hit suffers no penalty. A miss causes a full calculation, cache update, and memory-access retry.
- NB: a memory-access retry is different than an interrupt!
- The pseudocode also checks for access permission. BTW: lines 15 and 16 appear redundant with lines 4 and 9.

Who Handles The TLB Miss? (1 of 2)

- Upon a TLB miss, the linear or multi-level page table must be accessed, to calculate a translation.
- This can be done in hardware, by the MMU, expecting a vendor-specified page-table format.
- An older design, or CISC CPU (e.g., x86-64), likely does this.

Who Handles The TLB Miss? (2 of 2)

- For a simpler CPU, an MMU can simply trigger a page fault, to be handled by the CPU, *during* the current instruction's memory access. Before this access, or instruction, can be retried, a page-fault handler accesses the page table calculating the translation.
  It runs in kernel mode, perhaps bypassing the MMU, thereby using only physical addresses, avoiding further page faults.
  Or, to allow page-table access via virtual addresses (enjoying virtual benefits), those translations can be "wired" into the TLB, again avoiding further TLB misses.
- In either case, the TLB entry is updated.

TLB Contents: What's in There?

- As discussed earlier, associative memory doesn't really have addresses. For a TLB, each cell contains a VPN, a PFN, and a mask of bits.
- As with in-MMU segmentation registers, and in-memory page-table entries, several TLB-entry bits hold metadata for the VPN's virtual page.
- One marks an entry as *valid*; others, whether its virtual page:
  - has been read from (i.e., *referenced*)
  - has been written to (i.e., *dirty*)
  - *has* a physical frame (i.e., *present*)
- A entry also has virtual-page protection bits: read-only, read/write, executable.

Issue: Replacement Policy

- Typically, a context switch causes the TLB to be flushed, to prepare for a new page table, belonging to the next process. (Some systems try to avoid this overhead, by having processes share the TLB.)
- However, the TLB might become full, prior to a context switch. Which entry should be replaced by a new translation?
- A TLB entry's metadata bits can help us decide. This is so similar to deciding which physical page to swap-out to disk, when memory becomes full, that we defer.

The Big Picture

- A CPU relies on its MMU to perform virtual-to-physical memory-address translation. A TLB, other caches, main memory, and disk are also involved:

    `pub/ch19/BigPic.pdf`

“All I wanted to do was run a program...”

- When `exec` is called to execute a program, its process is already running.
- With on-demand paging, `exec` can start execution by creating page-table entries, but without reading the program’s file into physical frames. Those entries are marked as *valid* but not *present*, and what would be the PFN points to the program’s file (and an offset).
- When the process accesses one of those pages, the MMU page faults, the handler obtains a free page, and fills it with (part of) the program’s file from disk.
- Since an executable page is read-only, it can be swapped out by simply marking its PTE as not *present*.

Paging: Smaller Tables

- A linear page table contains an entry for each VPN. For 256TB of virtual memory, 4KB pages, and 4GB of physical memory (32-bit entries), each table would need 256 gigabytes of memory:

$$\frac{2^{48}}{2^{12}} \times 2^2 = 2^{38} = 256\text{GB}$$

- Each process has a page table. For 128 processes, page tables would need 32 terabytes of memory:

$$2^{38} \times 2^7 = 2^{45} = 32\text{TB}$$

- That's too much!

Idea #0: Page Size

- We could double or quadruple the size of each page, reducing the number of entries in the table.
- However, that would increase internal fragmentation.
- And, where do you stop? Having one gigantic page isn't exactly paging.
- Perhaps the number of pages should be the number of page-table entries that fit in one page.

Idea #1: Hybrid

- With a large virtual address space, almost all of the table entries are not valid, not mapping to a physical page.
- We could avoid storing most of those entries, with several small page tables.
- Rather than an MMU having just one page-table base register, we could have several, as with segmentation.
- Each could contain the base address of a small page table, one for the process's code, static, stack, and heap segments (resp.). Some tables might only have one entry. Bounds registers could hold sizes.
- This increases state, for a context switch. It assumes a process uses those segments. It may increase external fragmentation.

Idea #2: Tree (1 of 2)

- An array is indexable but static (fast). A tree is traversable and dynamic (small).
- We could combine them, to build a sparse page table, called a *multi-level* page table.
- Two or three levels suffice, for a logarithmically small table.

Idea #2: Tree (2 of 2)

- Suppose a page can hold N physical addresses (e.g., 1K 4-byte addresses).
- The MMU's page-table base register could hold the address of the "directory" page table, which holds the addresses of N ordinary page tables. You can imagine a second, or third, directory level.
- $lg(N)$ high bits of a VPN could index the directory. If the indexed address is valid (e.g., nonzero), the VPN's lower bits could index the ordinary page table at that address. Again, you can imagine multiple directory levels.
- Space savings accrue from sparse (wrt. validity) directory page table(s). Consider the big gap between heap and stack segments.

Chips and Disks

- x86-64 uses four-level paging; its MMU walks the tree.
  ARMv8 uses three-level paging; its MMU walks the tree.
  MIPS uses segmentation plus paging.
- Here's a good picture of an x86-64 tree, from `https://os.phil-opp.com`:
  `pub/ch20/pt-tree.pdf`
- Another idea is to swap-out page-table pages, to disk, just like pages. We'll defer.

Beyond Physical Memory: Mechanisms (1 of 2)

- We've seen how an MMU's address-translation hardware allows an OS to present, to each of a set of virtual CPUs, the illusion of a large and uniform memory, backed by physical memory.
- Each process's virtual-address space is typically *much* larger than the physical-address space populated by actual memory chips.
- What happens when a process accesses a new page, but no free frame exists? What happens when a workload's page count exceeds the computer's frame count? What happens when memory is "full"?

Beyond Physical Memory: Mechanisms (2 of 2)

- When the storage capacity at one level of a hierarchy is exhausted, data is stored at the next level: in larger, slower, and cheaper storage.
- To wit: registers, L1 cache, L2 cache, L3 cache, main memory, disk-drive cache, and disk. Even main-memory modules might have caches.
- Here, data in a physical-memory frame is *swapped* to disk, freeing the frame for another purpose. Of course, the page table must be updated, to indicate this.
- Eventually, that data is *swapped* from disk, into a different frame, and the page table is updated.

Swap (1 of 4)

- The kernel, perhaps with the help of a user-space process, needs one or more disk areas to store frame data, called *swap space*.
- Each can be a disk partition, containing a swap filesystem, or a plain file in an ordinary filesystem. Each is preallocated, preformatted, and then activated (e.g., via `dd`, `mkswap`, and `swapon`).
- Recommended swap-space size is about the size of physical memory.

Swap (2 of 4)

Without swap, we have seen that each virtual-memory access does this:

1. Split virtual address: VPN+offset.
2. Test TLB[VPN] *valid*:
   (a) TLB Hit: The entry's protection is enforced. Its PFN+offset is the physical address.
   (b) TLB Miss: Test PT[VPN] *valid*:
      i. Page Miss: VPN invalid: no frame. Kill process.
      ii. Page Hit: Update TLB. Retry access.

Swap (3 of 4)

With swap, part 2(b)ii becomes more complex. At this point, the VPN has a *valid* PTE. At one time, it had a frame, but its data may no longer be *present* in a frame. It may have been swapped to disk. So, part 2(b)ii becomes:

1. Test PTE *present*:
   (a) Page hit: Update TLB: PFN and *valid*.
   (b) Page fault: Let NEW be a free frame (next). Update TLB[VPN]: NEW and *valid* (optional). Update PTE: NEW and *present*. Read NEW from disk.
2. Retry access.

Swap (4 of 4)

The kernel maintains a nonempty set of free frames. As the number of free frames falls below a threshold, used frames are written to disk, freeing them:

1. Choose VPN, a virtual page to swap out. Let PTE=PT[VPN] be its frame. PTE is *valid* and *present*.
2. Update TLB[VPN]: not *valid*.
3. Update PTE: not *present*.
4. Write PTE to disk.

The Page Fault

- A virtual-to-physical memory-address translation begins in hardware.
- Upon TLB miss, some MMUs use hardware to continue translation, accessing a page table; others continue with software.
- When the page-table entry is *valid*, but not *present* (i.e., it's been swapped out), all MMUs resort to software to finish the translation. In other words, a page fault is serviced by a page-fault handler, which is similar to an interrupt handler.
- A page fault is serviced by software, because accessing swap space is a slow disk operation. It will block, during I/O. Plus, an MMU doesn't know anything about disks.

When Replacements Really Occur

- Earlier, we assumed that the kernel maintained a nonempty set of free frames.
- How many frames are free? When is a frame freed? Which frame is freed?
- The kernel keeps the number of free frames between a *low watermark* and *high watermark*. These limits are checked and maintained, periodically (e.g., by Linux's `kswapd`). Spare frames are important, because freeing frames requires memory.
- Obviously, a process's frames are freed as it terminates (unless shared).
- More interestingly, a frame can be freed by writing its data to swap space. A frame is chosen for this treatment, by a page-replacement policy (next chapter).

Beyond Physical Memory: Policies

- When the kernel wants to free a frame, it selects a page-table entry that is *valid* and *present*, and *evicts* its page.
- Eviction changes a PTE to not *present*, and (maybe) writes its frame's content to swap space.
- There are several policies for selecting a page to evict. They operate on a system-wide basis, ignoring process ownership.
- The overall goal is to evict a page that won't be used anymore, or will be used the furthest in the future. This unachievable goal is (hypothetically) the result of the *Optimal* replacement policy.
- A policy might also try to avoid evicting a *dirty* page: a modified one. It's data would need to be written to swap space, which is costly.

A Simple Policy: FIFO

- This policy evicts the page that has been present the longest.
- Unfortunately, this page might be the most popular.

Another Simple Policy: Random

- This one evicts a randomly chosen page.
- Unfortunately, page access is not random.
- Our time-tested *Principle of Locality* suggests that processes tend to exhibit *spatial* and *temporal* locality.
- Spatial locality says that a memory access at a particular address increases the chance of an access at a nearby address.
- Temporal locality says that a memory access at a particular address is likely to be repeated, soon.

Using History: LRU

- The success of a page-replacement policy depends upon future events. Some policies hope to predict what will happen, based upon what has happened, and faith in locality.
- *Least Frequently/Recently Used* (LFU and LRU) are two such policies. Linux implements LRU.
- Both policies store a compact summary of access history for each page. LFU keeps an access count. LRU keeps a periodically zeroed access indicator.
- Of course, if pages are accessed in a round-robin way, an “LRU” page might be accessed next.
- Apparently, policies that replace “Least” with “Most” have been proposed. MRU would do well with a round-robin workload.

Thrashing

- Locality suggests that each process has a *working set*: those pages it is actively accessing. A page-replacement policy tries to keep working sets present, in memory.
- An optimist might hope that all this clever paging and swapping would allow a system with “oversubscribed memory,” or “memory pressure,” to continue processing, albeit with linear performance degradation.
- Sadly, as paging/swapping overhead climbs, throughput quickly drops to zero: a phenomenon called *thrashing*.
- Thrashing once had a physical manifestation. The cabinet containing the swap disk might vibrate like an unbalanced washing machine.

Concurrency: An Introduction (1 of 2)

- We have seen that each process runs on its own virtual CPU, time-multiplexing the physical CPU.
- We have seen each process has its own virtual memory, sharing physical memory. Every process's virtual address space has the same range of virtual addresses, perhaps starting at zero.
- Recall, also, that each process divides its virtual memory into segments, containing its: code, static data, stack, and heap.
- A process runs under the illusion that it is the *only* process, with its own CPU and memory.

Concurrency: An Introduction (2 of 2)

- Now, we relax this contraint. We allow *multiple* “processes” to run on a virtual CPU and share a virtual address space, but now we call them *threads*.
- Although they share a virtual CPU and memory, each thread has its own register context (e.g., PC) and stack segment (e.g., for arguments, local variables, and return addresses).
- This allows them to execute concurrently, either interleaved or on separate cores.

Why Use Threads? (1 of 2)

- Multiprocessor computers have been around for a while, in various forms, but only for special applications.
- Multicore processors are typical, now (e.g., in your phone).
- Originally, parallelizing compilers could generate code for multiprocessors. They struggled to identify opportunities for concurrency.
- Humans are better than compilers at designing concurrent algorithms or software.
- Later, parallel/concurrent programming languages allowed programmers to do so (e.g., Concurrent Pascal and Go).

Why Use Threads? (2 of 2)

- Now, any language with a threading library can take advantage of a multicore processor, with explicit function calls to manage threads.
- Even on a single-processor single-core computer, thread interleaving is valuable.
- Imagine a multithreaded server. After a thread begins servicing one request, and blocks on I/O, another thread can immediately begin servicing a second request. The server is more responsive.
- Processes make sense for heavyweight tasks. Separate memory provides isolation and security from interference and leaks.
- Threads make sense for lightweight tasks. Shared memory supports cooperation and communication.

An Example: Thread Creation

- There have been many thread libraries. Some are user-space systems, and some rely on the OS kernel.
- We will use the popular and portable kernel-threads system, called POSIX Threads (pthreads).
- Here is a simple pthreads example:

  `pub/ch26/ThreadSync/ThreadSync.c`
- Notice how the threads safely share a variable.

Why It Gets Worse: Shared Data

- As our textbook says:

  Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it simply gets worse. Much worse.
- Here is a trickier example:

  `pub/ch26/ThreadCount/ThreadCount.c`
- Notice how the threads unsafely share a variable.

The Heart Of The Problem: Uncontrolled Scheduling (1 of 3)

- Remember, threads share memory (e.g., the variable `counter`), but each has its own register context. With a single-core processor, context switches accompany interleaving. With a multicore processor, each core has its own registers.
- Interleaving can occur between any two machine instructions. A preempting timer interrupt is deferred until after the current instruction.
- Different cores execute instructions independently. The cores and clocks may even be different from one another.
- So, we must focus on machine code.

The Heart Of The Problem: Uncontrolled Scheduling (2 of 3)

- To increment `counter`, its value is moved from memory into a register, incremented, and then moved back to memory. Ordinarily, a value in memory is not incremented *in situ*.
- We can disassemble the executable file, with `objdump -dS`.
- Or, we can look at the generated assembly file, with `gcc -S`.
- Now, we see the problem. Multiple threads change *their* register value. The last thread to move it to memory "wins."
- What about `counter++`? Does it matter?

The Heart Of The Problem: Uncontrolled Scheduling (3 of 3)

- This is called a *race condition* or, more specifically, a *data race*.
- It leads to *nondeterministic* or *indeterminate* results.
- The increment of `counter` is called a *critical section*. We need to execute it with *mutual exclusion*.
- All of this was pioneered by Edsger Dijkstra, in 1968, in his paper "Cooperating Sequential Processes." He was the first person to win the Turing Award not from the US or UK (he was Dutch), for this and other work.

The Wish For Atomicity (1 of 2)

- A thread could disable interrupts, while incrementing `counter`. Talk about brute force! Also, that would require two system calls.
- We could ask Intel to build us a CPU with an instruction that increments memory (q.v., `xadd`).
- We could use a microprogrammable CPU, and code our own memory-increment instruction.
- Be serious! Also, we might need other "super instructions."

The Wish For Atomicity (2 of 2)

- We could use/develop a high-level programming-language construct, or library, like:

```
mutex {
    counter=counter+1;
}
```

or:

```
mutex_begin();
counter=counter+1;
mutex_end();
```

However, this may prevent acceptable concurrency, or may not be flexible enough.

One More Problem: Waiting For Another Thread

- We've seen that threads need to share variables, and execute critical sections with mutual exclusion.
- In addition, threads sometimes need to wait for one another. In the next chapter, we'll see how to do this, with *condition variables*.
- Why are we studying all this concurrency and thread stuff in an Operating Systems course? Only the kernel has permission and resources to implement threads effectively. The kernel *is* multithreaded software (at least, Linux is).

Interlude: Thread API

- In this chapter we'll see how to create a thread, pass it an argument, and wait for it to finish.
- We'll also see locks and condition variables.
- Finally, we'll see how to compile and link a multithreaded program.

Thread Creation (1 of 3)

- We saw an example of this, last chapter:

  `pub/ch26/ThreadSync/ThreadSync.c`
- The main thread calls `pthread_create` to create a new thread:

```
pthread_create(&tids[i],
               0,
               count,
               (void *)(long)i);
```

- The first argument sort-of returns the new thread's identifer (TID). You usually want to wait for the thread to finish, so save its TID.

Thread Creation (2 of 3)

- The second argument sets the attributes of the new thread. For defaults, pass 0. But, for example, to create a “detached” (i.e., non-joinable) thread, which doesn’t need to be waited for, pass an appropriate `attr` argument:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_DETACHED);
...
pthread_attr_destroy(&attr);
```

Thread Creation (3 of 3)

- The third argument is a pointer to the function the new thread should begin executing. Recall that each thread has its own stack. The function takes one `void*` argument, and returns a `void*`.
- The fourth argument is the value to pass to the thread-starting function as its argument. If you need to pass multiple "arguments," pass a pointer to an aggregate value (e.g., a `struct`).

Thread Completion (1 of 2)

- Back to our thread example:

  pub/ch26/ThreadSync/ThreadSync.c

- The main thread, or another thread, calls `pthread_join` to wait for a joinable thread to finish:

```
pthread_join(tids[i],0);
```

- The first argument is the TID of the thread to wait for. If that thread has already finished, `pthread_join` returns immediately.
- A thread finishes by returning from the function that it called as it was created, or by calling `pthread_exit`. In either case, it can “return” a `void*` value to the thread calling `pthread_join`.

Thread Completion (2 of 2)

- The second argument, if not 0, is a pointer to the finished thread's `void*` return value. Of course, that value cannot point to one of the thread's local variables, because the thread's stack is deallocated as the thread finishes.
- A joinable thread must be joined, or it becomes a "zombie" thread, which retains resources even though it has finished.

Locks (1 of 2)

- Recall our tricky example, where the threads unsafely shared a variable:

  `pub/ch26/ThreadCount/ThreadCount.c`
- A critical section can be executed with mutual exclusion, with a *lock*:

  `pub/ch27/ThreadLock1/ThreadLock1.c`
- This is better, but it isn't quite right. Why does it sometimes print 10 and 11? There are still accesses to `counter` outside of the locked section.
- How can we fix it? Enlarge the critical section, but be careful:

  `pub/ch27/ThreadLock2/ThreadLock2.c`

Locks (2 of 2)

- When a thread tries to lock a locked pthreads mutex, the thread sleeps until it acquires the mutex.
- In contrast, when a thread tries to lock a locked pthreads *spin lock*, the thread loops until it acquires the spin lock.

Condition Variables (1 of 2)

- Often, a thread doesn't need to acquire a mutex, until a particular condition holds. The condition is eventually made true by some other thread. Repeatedly locking and unlocking the mutex, just to test the condition, is wasteful of resources.
- A *condition variable* allows a thread that makes the condition true to signal one or more threads that are waiting for the condition to hold.

Condition Variables (2 of 2)

- For example, suppose we have a data structure (e.g., a `Box`) that holds a value, read and written by multiple threads:

  `pub/ch26/BoxMon/BoxMon.c`

  `pub/ch26/BoxMon/Box.c`
- A lock provides mutual exclusion for a condition variable. The wait frees the lock, but when the wait returns, the lock is reacquired.
- Notice the condition-testing loops. From the `man` page:

  Spurious wakeups from the `pthread_cond_wait` *function may occur.*

  Furthermore, the signaling thread may have called `pthread_cond_broadcast`, and some other signaled thread has made the condition false, again.

Locks

- This is a long chapter (20 pages). We've already seen how to *use* pthreads mutexes as locks, and I've mentioned pthreads spin locks.
- This chapter explains how to *implement* locks, in various ways.
- Eventually, we will see that Linux implements pthreads with the help of a system call named `futex`. The "f" is for "fast."

Locks: The Basic Idea (1 of 2)

- Suppose this is our one-and-only, and trivial, critical section:

```
balance=balance+1;
```

- We can use a (hypothetical) lock, for mutual exclusion, like this:

```
Lock mutex;
...
lock(&mutex);
balance=balance+1;
unlock(&mutex);
```

Locks: The Basic Idea (2 of 2)

- A lock is just a variable (e.g., a `struct` or pointer). In some way, it holds the state of the lock, its *owner* thread (if locked), and a queue of waiting threads.
- When a thread calls `lock` on an unlocked lock, the lock becomes locked, and the thread becomes the owner. Else, the thread is enqueued and becomes *Blocked*.
- When a lock's owner calls `unlock` on the lock, it is unlocked. If its queue is nonempty, a thread is dequeued, becomes the new owner, and becomes *Ready*.

Pthread Locks

- As we saw before, a pthreads lock is called a mutex:

```
#include <pthread.h>
...
pthread_mutex_t lock;
...
pthread_mutex_init(&lock,0);
...
pthread_mutex_lock(&lock);
balance=balance+1;
pthread_mutex_unlock(&lock);
...
pthread_mutex_destroy(&lock);
```

- Notice that the lock is passed to the various functions. This allows multiple locks, for fine-grained concurrency control.

Lock Implementation (1 of 2)

- A robust lock, which cannot be subverted by other threads, requires kernel and hardware support.
- At minimum, we need a machine instruction that can do what we would otherwise need two instructions to do, atomically (i.e., without an intervening preemption). We'll see several such instructions.
- And, since a thread might block, waiting to acquire a lock, the kernel's scheduler is involved.
- And, since part of the lock is allocated in kernel space, rather than within a thread's virtual address space, a system call is required to access it.

Lock Implementation (2 of 2)

- A good lock should certainly provide mutual exclusion.
- Threads waiting on a lock should enjoy fairness, and not suffer starvation.
- Lock operations should be fast. Indeed, performance is one reason to develop a multithreaded program.

Controlling Interrupts (1 of 2)

- Our textbook assumes a single-core processor, and gives this example:

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- We could try to “improve” it, allowing multiple locks and more concurrency, but it is still lame:

  `pub/ch28/lock.c`

  What would happen with multiple cores? A thread that disables interrupts only does so on its core.

Controlling Interrupts (2 of 2)

- There are several problems with the interrupt plan.
- We would need to decide *which* interrupts to disable,
- Interrupt-related instructions can only be executed in kernel mode. System calls would be required.
- Interrupts would need to be disabled for all cores and all processors (i.e., sockets). The latter is not possible.
- Disabling interrupts risks losing I/O (e.g., data read from a disk).
- Therefore, we abandon the interrupt plan.

A Failed Attempt: Just Using Loads/Stores

- Let's look more closely at our previous idea, after removing interrupt stuff, and analyze why it is incorrect:

```
1  void lock(int *lck) {
2    while (*lck)
3      ; // spin-wait
4    *lck=1;
5  }
6  void unlock(int *lck) { *lck=0; }
```

- Suppose there are two threads, which have both just executed line 2, and found that the lock's value was zero.
- So, both threads skip the loop body (line 3), and can execute line 4.
- After both threads have executed line 4, they both (think they) own the lock, and mutual exclusion has been violated.
- Also, if a thread is repeatedly executing line 3, it is wasting CPU resources. It should sleep.

Building Working Spin Locks with Test-And-Set (1 of 3)

- BTW: What happens if two threads assign to the same variable at “exactly” the same time?
- This sort of race condition can be avoided, with a single instruction, or dependent pair of instructions, that performs two operations, atomically. They have various names, and appeared in the early 1960’s.
- We will call it the *test-and-set* instruction. Atomic C version:

```c
int TestAndSet(int *lck) {
  int old=*lck;
  *lck=1;
  return old;
}
```

Building Working Spin Locks with Test-And-Set (2 of 3)

- It can be used to improve our lock:

```
void lock(int *lck) {
  while (TestAndSet(lck))
    ;  // spin-wait
}
void unlock(int *lck) { *lck=0; }
```

- Instructions in popular architectures:
  - Intel x86: `lock/bts`: locked bit-test-and-set.
  - SPARC: `ldstub`: load-store (unsigned byte).
  - ARM: `ldrexb/strexb`: load/store-register-exclusive (byte).
  - MIPS: `ll/sc`: load-linked and store-conditional.

Building Working Spin Locks with Test-And-Set (3 of 3)

- Recall our Chapter 27 solution, with the enlarged critical section:

  `pub/ch27/ThreadLock2/ThreadLock2.c`

- Here's an adaptation of it, with our Intel x86 locks:

  `pub/ch28/ThreadLock/ThreadLock.c`

  `pub/ch28/Lock/x86/TestAndSet.c`

- This program uses GCC's inline-assembly feature. It's arcane, but works pretty well.

Evaluating Spin Locks

- With test-and-set instructions, our spin locks are now correct: They guarantee mutual exclusion, even on a multicore processor.
- Regarding fairness: The thread that acquires the lock is determined by chance. With concurrency, we must assume the worst: an unlucky thread starves.
- Regarding performance:
  - A spinning thread wastes resources. It is using a core and repeatedly accessing memory, when it should sleep. On the plus side, the code and lock are likely in a cache.
  - On a single-core processor, this slows the other threads. Indeed, the lock can't be released until the spinning thread is preempted!

Compare-And-Swap

- Intel x86 also provides a more complex compare-and-exchange instruction.
- We can swap it with test-and-set, in our last example:

  `pub/ch28/ThreadLock/GNUmakefile`

  `.../x86/CompareAndExchange.c`

Load-Linked and Store-Conditional

- ARMv8 provides two dependent instructions. From Wikipedia:

  Load-link returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link.

- We can swap it with test-and-set, in our running example, and try it on a Raspberry Pi 3 Model A, with a Cortex-A53 (ARMv8) 64-bit SoC:

  `pub/ch28/ThreadLock/pi.jpg`
  `pub/ch28/ThreadLock/GNUmakefile`
  `.../aarch64/LoadLinkedAndStoreConditional.c`
  `pub/ch28/ThreadLock/pi.txt`

Fetch-And-Add

- Intel x86 and ARMv8 also provides a (more complex) exchange-and-add instruction.
- We can swap it with test-and-set, in our running example:

  `pub/ch28/ThreadLock/GNUmakefile`

  `.../x86/ExchangeAndAdd.c`

  `.../aarch64/ExchangeAndAdd.c`
- Notice that this instruction causes `lock` to *always* change the lock, even when it was already locked. That's why `unlock` decrements, rather than zeroes.

Too Much Spinning: What Now?

- Our several locking approaches all wait by spinning (i.e., looping). This is wasteful, especially on a single-core processor.
- One solution is for a thread that cannot acquire the lock to voluntarily *yield*, by calling a function like `yield`, thereby letting another thread run on the core.
- A `yield` call simply changes the thread's state from *Running* to *Ready*. Eventually, the scheduler lets the thread run, again, and it retries acquiring the lock.
- We can easily convert our running example to use the Linux version of `yield`:

  `pub/ch28/ThreadLock/GNUmakefile`

  `pub/ch28/Lock/Lock.c`
- One problem with this approach is that it is silly to wake a waiting thread, until the lock is free. Also, fairness is still ignored.

Using Queues: Sleeping Instead Of Spinning (1 of 3)

- When multiple threads have called `yield`, the scheduler decides which to wake next, and when to do so. We want more control over that decision.
- There is no reason to wake a waiting thread, until the thread holding the lock calls `unlock`.
- So far, all `unlock` does is zero the lock:

```
extern void unlock(Lock lck) { *lck=0; }
```

Using Queues: Sleeping Instead Of Spinning (2 of 3)

- However, suppose `lock` does this:
  1. Enqueue the thread onto the lock's queue.
  2. Call `wait` on that thread, rather than `yield`. This changes the thread's state from *Running* to *Blocked*.

  Some other *Ready* thread runs.
- Further, suppose `unlock` does this:
  1. Dequeue a thread from the lock's queue.
  2. Call `wake` on that thread. This changes that thread's state from *Blocked* to *Ready*.

  The *Running* thread continues to run.

Using Queues: Sleeping Instead Of Spinning (3 of 3)

- Now, a thread that can't acquire a lock blocks, until the lock is free, and a lock's sleeping threads are unblocked fairly.
- Sadly, Linux, and other Unix systems, don't provide `wait/wake`. Nor does pthreads. Sun's Solaris does: they are named `park/unpark`.
- Our textbook shows a lock implemented with `park/unpark`.

Different OS, Different Support

- However, Linux provides the futex (Fast User-space muTual EXclusion), which allows us to do this:

  `pub/ch28/ThreadLock/GNUmakefile`

  `pub/ch28/Lock/Lock.c`

  `pub/ch28/Lock/Wait.c`
- The `man` page for `futex` is very challenging.
- Our lock, and atomic lock/unlock code, is in user space. The waiting/queueing machinery is in kernel space. Hence, the system call.
- Yet, the kernel performs an atomic read-only access of the user-space lock! (To avoid race conditions.)
- On Linux, pthreads is atop futexes.

Lock-based Concurrent Data Structures

- We've seen how to *use* pthreads mutexes as locks, to ensure mutually exclusive access to critical sections of code.
- We've also seen how to *implement* locks, with special machine instructions.
- This chapter explains how to *use* locks, to allow multiple threads to share a data structure (e.g., a list), without interferring with one another, making the data structure *thread safe*.
- As with disabling interrupts, large-grain locking can reduce performance. So, we focus our attention on fine-grain locks, protecting small critical sections.

Concurrent Counters (1 of 3)

- We've seen examples of multiple threads updating a counter.
- We can think of our counter as a object-oriented (OO) data structure. It has one (private) instance variable, a constructor, and some (public) methods (e.g., `incr` and `get`).
- We can make it thread safe:
  - Add a mutex instance variable, initialized by the constructor.
  - Bracket each method body with lock/unlock calls.
- This is a *monitor*. Java makes this trivial, with its `synchronized` keyword (Java's only concurrency feature).

Concurrent Counters (2 of 3)

- If your newly thread-safe data structure is fast enough, advance to the next chapter.
- However, the overhead of this brute-force thread safety may slow counting by a factor of 100. We may have hoped concurrency would make our program faster, but we've made it slower. Adding more threads makes it worse. Our solution does not *scale*.
- These sorts of problems have received much study. There are many clever, albeit *ad-hoc*, improvements.

Concurrent Counters (3 of 3)

- Here, for example:
  - We can give the threads on each CPU a local, but still thread-safe, (approximate) counter. This avoids contention between threads on different CPUs.
  - Periodically, a local counter updates the global counter, synchronized by its global lock. The local counter is then zeroed.
  - The global counter's value is correct when the program begins and ends, but tunably approximate during execution, or when updates are forced. That might be good enough.

Concurrent Linked Lists and Queues (1 of 2)

- Containers or collections are common data structures: sets, lists, trees, tables, etc. They are easy to wrap, as monitors.
- To improve performance, critical sections can be minimized, by analyzing the code.
- For example, when adding an item to the front of a linked list, the node-creation code need not be in the critical section. The “shared” part is the link manipulation.
- This also prevents forgetting to unlock the mutex, if node-creation fails (assuming the program doesn’t terminate, when that happens).

Concurrent Linked Lists and Queues (2 of 2)

- With a queue, a mutex for each end can decouple enqueue and dequeue operations. The corner cases of an empty or one-element queue can be finessed with “dummy” nodes, a common data-structure trick.
- You could even use a mutex for *each* node in the structure.

Concurrent Hash Tables

- Hash tables are popular for implementing containers for large numbers of items. Thus, they might be concurrenly accessed by large numbers of threads. Thus, performance attention make sense.
- A common structure to hold the items in each bucket is a linked list. We can simply reuse our previous ideas, for each bucket.
- Some hash tables resize themselves, when they become full. A whole-table mutex can prevent interference during such an operation.

Condition Variables

- We saw how thread A could react to a lock being busy by waiting (i.e., sleeping), until thread B awakened it. Thread B wouldn't do so unless the lock was free.
- We used a futex to put A to sleep.
- Upon awakening, A still had to try to acquire the lock, because there was no guarantee that the lock was still free. Some other thread might have locked it!
- Often, A does not just want to acquire the lock. A might also want some other condition to be true (e.g., a resource to be available).
- We could add more logic to our futex solution, but this is such a common scenario that pthreads provides a prepackaged solution: a *condition variable*.

Definition and Routines (1 of 2)

- We already saw this, in Chapter 26 and the Whack-a-Mole homework, but an example will review the pthreads condition-variable API.
- Suppose our multithreaded counting program employed threads to not only increment the counter, but also decrement the counter. Further, suppose we wanted to bracket the counter's value between negative and positive limits. It now runs forever:

  `pub/ch30/ThreadLock/ThreadLock.c`
- Some arithmetic voodoo unifies the increment and decrement cases.

Definition and Routines (2 of 2)

- A thread must acquire the lock to check, and perhaps wait on, its condition. If it starts to wait, the lock is freed; when it wakes, it again holds the lock, and *must* recheck its condition.
- After the critical section, of course, the thread must free the lock.
- If our thread has changed the state of its process (e.g., a variable's value), such that one or more conditions *other* threads might be waiting on might have become true, our thread should signal those condition variables, perhaps waking those other threads.

The Producer/Consumer (Bounded Buffer) Problem (1 of 2)

- Our previous example, concurrently incrementing/decrementing a counter, fully demonstrates condition variables. The shared resource is the 32-bit memory location holding the counter's value.
- However, the *classical* example is the ubiquitous Producer/Consumer Problem, studied by Edsger Dijkstra, circa 1972. For example:

```
$ ls -1 | wc -l
```

  Here, the kernel manages a pipe: the shared resource that allows these two concurrent processes to communicate. A pipe has a large, but bounded, capacity. By default, reading from an empty pipe, or writing to a full pipe, blocks.

The Producer/Consumer (Bounded Buffer) Problem (2 of 2)

- Dijkstra solved it with something called a semaphore, which we'll get to, but condition variables work pretty well.
- We can replace our counter with a buffer:

  `pub/ch30/ProdCons/buffer.h`

  `pub/ch30/ProdCons/buffer.c`

  `pub/ch30/ProdCons/ProdCons.c`
- NB: The "work" of production and consumption is done *outside* of the critical section.
- NB: The `while (busy())` loop may seem unneeded, but it *is* needed. The application may have performed a broadcast signal, and/or pthreads threatens "spurious wakeups."

Covering Conditions (1 of 2)

- Suppose an executing thread T has made some condition $C = C_A \vee C_B$ true, by making one of C_A or C_B true, and therefore signals the condition variable V_C.
- Further, suppose two other threads, A and B, are waiting on V_C, but A only cares about C_A and B only cares about C_B.
- A or B may wake up to find that the subcondition it cares about is still false. Should it re-signal V_C before re-sleeping? When does this end? Might such a thread receive its own signal?
- C is called a *covering condition*. A better plan might be for T to *broadcast* the signal, with `pthread_cond_broadcast`.

Covering Conditions (2 of 2)

- Try to avoid broadcasting, especially to work around a design flaw.
- A better idea is to factor the condition: use two condition variables V_A and V_B, and have A wait on V_A and B wait on V_B.
- In this way, when a thread wakes up, there is a better chance that the condition it cares about is true.
- You still need a loop, though! See:

  `pub/ch30/ThreadLock/ThreadLock.c`

  `pub/ch30/ProdCons/ProdCons.c`
- The third argument in a `FUTEX_WAKE` call can cause broadcasting:

  `pub/ch28/Lock/Wait.c`

Semaphores

- We've seen locks, mutexes, futexes, and condition variables.
- Circa 1970, Edsger Dijkstra invented, and solved his concurrency problems with, what he called a *semaphore*.
- A semaphore is sort-of a combination of those other synchronization primitives. It's all you need.

Semaphores: A Definition (1 of 2)

- The Portable Operating System Interface (POSIX), provides two semaphore functions: `sem_wait` and `sem_post`. Pthreads doesn't provide semaphores, but may use them.
- Dijkstra was Dutch, and named the functions P and V, initials of Dutch words. For us:

P:	wait	down	acquire	pend	procure
V:	post	up	release	signal	vacate

I like the last pair.

Semaphores: A Definition (2 of 2)

- Their semantics seem simple enough:
  - `int sem_init(sem_t *s, int pshared, int value)`: Initialize the semaphore's value, the number of "resources." For threads, `pshared` is zero.
  - `int sem_wait(sem_t *s)`: Decrement the semaphore's value. If the new value is negative, wait.
  - `int sem_post(sem_t *s)`: Increment the semaphore's value. If threads are waiting, wake one.
- They return zero upon success.

Binary Semaphores (Locks)
and
Semaphores For Ordering

- A semaphore with values of only zero and one is called a *binary* semaphore. Basically, it's a lock.
- Compare:
  `pub/ch30/ThreadLock/ThreadLock.c`
  `pub/ch31/ThreadLock/ThreadLock.c`
- Notice how the condition, and code to test it, has been absorbed by the general semaphores.

The Producer/Consumer (Bounded Buffer) Problem

- Dijkstra solved the Producer/Consumer Problem with his semaphores.
- Compare:

  `pub/ch30/ProdCons/ProdCons.c`

  `pub/ch31/ProdCons/ProdCons.c`
- Again, notice how the conditions, and code to test them, have been absorbed by the general semaphores.
- Why would you ever use condition variables?
- Spurn the seductiveness of these seemingly simple solutions. Make sure you understand our textbook's evolution of a correct one.
- For example, what if the wait/post pairs were transposed? Deadlock! We'll discuss this again, later.

Reader-Writer Locks (1 of 2)

- We saw, a couple of chapters ago, how a lock could provide mutual exclusion, to make a data structure (e.g., a queue) thread safe.
- Sometimes, that's overkill. Often mutual exclusion is only required to *modify* a data structure; merely accessing (i.e., reading) it is already thread safe.
- A good way to understand this is:
  - A reader excludes writers.
  - A writer just excludes other writers.
- This is the Reader-Writer Problem. We can solve it with semaphores.

Reader-Writer Locks (2 of 2)

- Our example is a container holding a string of characters. Another enhancement transfers synchronization logic/data from clients into the container:

  `pub/ch31/ReadWrit/ReadWrit.c`
  `pub/ch31/ReadWrit/mtsb.h`
  `pub/ch31/ReadWrit/mtsb.c`
- Our semaphore solution makes tinkering with fairness difficult. Using `sem_trywait` and `sem_timedwait` could help.
- Pthreads provides reader/writer locks.

The Dining Philosophers (1 of 2)

- This problem, again from Djikstra, explores resource-contention *deadlock*: a state where all of a set of threads are waiting on conditions that can only occur when they are not all waiting. That is, they are all waiting on each other.
- Imagine philosophers at a table, each alternating between thinking and eating. A utensil is between each pair of philosophers. Historically, each is a fork, for spaghetti (that's silly). Realistically, each is a chopstick, for rice:

  `pub/ch31/Phils/Phils.c`

The Dining Philosophers (2 of 2)

- There are several approaches to deadlock, proactive and reactive:
  - A thread can request resources in an agreed upon order, preventing cycles.
  - A thread can return acquired resources, when a later request would cause a wait, retrying later.
  - A thread can request resource sets, all at once, perhaps from some sort of resource server.
  - A thread can request resources asynchronously, and be notified as each becomes available.
  - The OS can preempt a thread's acquisitions, perhaps causing a retry.
  - The OS can cancel a thread, requesting it to exit, perhaps causing it to free its resources.

Thread Throttling

- A semaphore can also be used to prevent too many threads from engaging in a resource-intensive activitiy, at once.
- Simply initialize a semaphore to the number of threads you want to allow, and have the threads wait/post that semaphore before/after that activity.

How To Implement Semaphores

- A semaphore can be implemented using a pthreads mutex and condition variable. Our textbook shows that.
- Or, we can use one of our locks. It can wait by spinning, yielding, or futexing.
- This example is our *unchanged* Producer/Consumer solution, with a homegrown semaphore, built atop test-and-set and yielding:

  `pub/ch31/Semaphore/semaphore.h`

  `pub/ch31/Semaphore/semaphore.c`

  This is confusing, partly because:
  - a thread sleeps on a zero semaphore
  - a thread sleeps on a nonzero lock

Common Concurrency Problems (1 of 2)

- Developing concurrent software, multithread and/or multiprocess programs, is challenging.
- Aside from the potential for deadlock, discussed earlier, several dangers lurk in the shadows.
- The root causes, of course, are multiple threads of control and communication, or interference, between threads. Regular chess becomes three-dimensional chess.

Common Concurrency Problems (2 of 2)

- A contributing cause is the non-determinancy injected by the thread and process schedulers. Bugs become spurious bugs (aka, *Heisenbugs*).
- These factors conspire against our development tools. GDB must allow a programmer to choose and follow particular threads. Valgrind must analyze thread-local stacks, as well as the shared heap. Frameworks and libraries must be thread-safe (aka, reentrant).

What Types Of Bugs Exist?

- Our textbook cites a study that classified concurrency bugs found in four popular applications (e.g., the Apache web server).
- About a third of the bugs were related to deadlock.
- The non-deadlock bugs can be further categorized.

Non-Deadlock Bugs (1 of 2)

- Atomicity-Violation Bugs: These are simply unrecognized critical sections.
- When a region of code requires mutual exclusion, for correctness, it must be protected by synchronization mechanisms (e.g., mutexes).

Non-Deadlock Bugs (2 of 2)

- Order-Violation Bugs: These are accesses to shared memory that must occur in a particular sequence, where that sequence is not enforced (e.g., a shared variable is accessed by other threads, before the main thread can initialize it).
- When an execution requires a certain order, for correctness, it must be constrained by synchronization mechanisms (e.g., condition variables). This has been called *barrier* synchronization.

Deadlock Bugs (1 of 2)

- All four of these conditions are required for a deadlock to occur:
  - Mutual exclusion: Resources can only be acquired by one thread at a time.
  - Hold-and-wait: Already acquired resources cannot be released, if additional resources cannot be acquired.
  - No preemption: Acquired resources cannot be forcibly taken away from threads holding them.
  - Circular wait: Threads can form a cyclic structure of waiting for one another to release resources.
- We discussed how to prevent these conditions, at the end of the last chapter. You only need to prevent one of them!

Deadlock Bugs (2 of 2)

- Our textbook also suggests: If a scheduler knows, in advance, the resource-request behavior of a set of threads, they can be scheduled in a way that precludes deadlock. This is unrealistic.
- Another, more realistic, approach is to detect that deadlock has occurred, and ruthlessly terminate the perpetrators (e.g., reboot). This falls into the category of: "Not everything worth doing is worth doing well."

Debugging Concurrency Problems (1 of 2)

- Our textbook does not discuss this, but I have found several techniques to be very helpful.
- While your application is executing, have it log detailed information about what it is doing. This should include a timestamp, PID, TID, and a descriptive message.
- Add `LOG(...)` “statements” to your code as you debug it, and *leave them in the code*.

Debugging Concurrency Problems (2 of 2)

- Turn logging on or off with a command-line option, not a macro that requires recompilation. Use other options to select the logging level and log file. The log file should be set to be unbuffered (ala, `stderr`).
- To reduce the amount of log data, use another command-line option to select the module(s) or source file(s) to log, via a pattern matched against the value of the `__FILE__` macro (which you can store in a variable, for run-time comparison).

I/O Devices

- A CPU can directly read or write memory, with a single machine instruction.
- Accessing an I/O device (e.g., a disk) requires multiple machine instructions, which direct some sort of controller, to read and/or write the device. The controller can be part of the device.
- Some systems address a device controller as if it is one or more memory locations (i.e., memory-mapped I/O). Others use special machine instructions, if the CPU instruction set has them, that give device controllers a separate address space.
- Device controllers, and I/O devices, use buses other than the main CPU/memory bus, called *I/O buses* (e.g., PCI) and/or *peripheral buses* (e.g., SATA or USB).

A Canonical Device

- A device controller, or the controller part of the device, can contain the following:
  - Several addressable “registers” are the interface to the CPU. Typically, a CPU writes to a *command register*, reads from a *status register*, and transfers data via a *data register*.
  - Device-specific electronics are the interface to the device itself (e.g., a WiFi or Bluetooth radio).
  - A little “computer,” with a little CPU, and various kinds of memory. Some of the memory holds firmware, which the little CPU executes.

The Canonical Protocol

- Our textbook over simplifies, claiming I/O occurs, via a device controller, with a protocol like this:

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
// (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device finishes request
```

- This style of spinning on the content of a status register is called *polling*.
- This style of a CPU writing directly to a data register is called *programmed I/O*.
- This *might* be how one byte is written to a device, but a one-byte read would be different. Furthermore, writing a block of data would require another loop.

Lowering CPU Overhead With Interrupts

- Polling wastes CPU cycles. Since I/O is slow, a better approach is for the process to sleep, so the CPU can context switch to a different process.
- When the device is done, it can raise a hardware interrupt, causing the CPU to (basically) call an *interrupt service routine* (aka, *interrupt handler*).
- The handler finishes the I/O request and wakes the sleeping process.
- This allows computation and I/O to *overlap*.

More Efficient Data Movement with DMA (1 of 2)

- However, if the device requires attention after each byte is transferred, the CPU will be interrupted too often.
- *Direct memory access* (DMA) solves this.
- A DMA controller might be a standalone device, shared by device controllers, or it might be integrated within a device controller (e.g., for a disk controller).
- Before I/O, the CPU sets up a DMA controller to transfer a block of data (address and size) to/from a particular device controller.

More Efficient Data Movement with DMA (2 of 2)

- After I/O begins, the process sleeps, and the CPU context switches.
- During I/O, the DMA controller read/writes main memory *while* the CPU executes other processes. Memory contention is arbitrated, or dual-port memory avoids contention.
- After I/O ends, one or both of the controllers interrupt the CPU.

Methods Of Device Interaction

- If a CPU instruction set has I/O instructions (e.g., Intel's `in` and `out`), they are privileged.
- Otherwise, memory-mapped I/O puts a device at particular memory addresses, which are protected like other memory segments.
- In either case, for a process to do I/O, it must make a system call.

Fitting Into The OS: The Device Driver

- The kernel's I/O system calls present an I/O abstraction to a process: `open`, `read`, `write`, `close`, etc.
- The abstraction is implemented, for different hardware devices, by appropriate *device drivers*.
- A device driver executes as part of the kernel, but can be loaded and unloaded as a *kernel module*.
- A process communicates with a device driver by accessing a "node" that looks like a file, often below `/dev`. See the `man` page for `mknod`.

Case Study: A Simple [IDE] Disk Driver

- Our textbook shows device driver code to control a disk controller, but it omits the kernel/driver interface code.
- Instead, let's omit the hardware code, and focus on the kernel/driver interface for an actual Linux driver module, which will prepare you for our homework assignment.
- A driver doesn't *have* to be a module. It can be compiled into the kernel. However, a module is more flexible, and doesn't bloat the kernel.

A Simple Driver (1 of 5)

- Recall that the kernel's I/O system calls present an I/O abstraction to a process: `open`, `read`, `write`, `close`, etc.
- The implementation of a device driver implements that abstraction:
  - It overrides some of these functions, by defining its own versions, and replacing corresponding entries in a table of function pointers. This is much like an OO subclass overriding superclass methods, by replacing entries in an object's vtable.
  - It allocates memory to hold state for these functions: for each driver, and for each instance of a driver. This is much like an OO class defining static variables and instance variables (resp.).

A Simple Driver (2 of 5)

- Consider a trivial Linux driver, which simply says "Hello world!" Usage:

  `pub/hw5/Hello/TryHello.c`
- The abstraction lets a client treat the driver like an ordinary file.
- Assuming the `Makefile` has already installed the driver's module, and created a node named `/dev/Hello` for it, `main` begins by opening the "file." Then, blocks of characters are read from it, and echo-ed to `stdout`, until end-of-file. Finally, the file is closed.
- Remember, the characters could be coming from some weird piece of hardware (e.g., a SETI telescope).

A Simple Driver (3 of 5)

- Here's the `Makefile`:

  `pub/hw5/Hello/Makefile`
- Much of it is boilerplate, interfacing with the kernel build system. I've added installation and testing targets.
- Here's the module/driver source file:

  `pub/hw5/Hello/Hello.c`
- Again, much of it is boilerplate.
- The `Device` structure corresponds to an OO class's static variables, as evidenced by the `device` variable.
- The `File` structure corresponds to an OO class's instance variables.

A Simple Driver (4 of 5)

- Function `open` is called, eventually, when a process calls `open`. Argument `inode` refers to the node `/dev/Hello`; `filp` refers to the process's local variable `fd`.
- Function `release` is called, eventually, when a process calls `close`. Don't leak kernel memory!
- Of course, the `read` functions are connected. Notice:
  - data is moved between kernel and user memory
  - the 0 byte is not moved
  - a subsequent `read` again starts at the beginning of the string

A Simple Driver (5 of 5)

- The `ops` structure specifies function overriding. A (super) driver's functions might be overridden by another (sub) driver's functions. Imagine a generic "disk" driver with IDE, SATA, and SCSI sub-drivers.
- When the module is installed (by `insmod`), `my_init` calls `cdev_init` to do the overriding.
- When the module is uninstalled (by `rmmod`), `my_exit` calls `cdev_del` to undo the overriding.

Hard Disk Drives

- This chapter provides much detail about disk hardware. We'll summarize briefly.
- Integrated disk controllers provide buffering, caching, and operation scheduling (with clever algorithms).
- A disk *mechanism* is a spindle, holding platters, with concentric *tracks* (forming *cylinders*), divided into *sectors* (of perhaps 512 bytes).
- Moving the read/write head(s) to a different track/cylinder takes much longer than waiting for a sector to revolve under the head.
- Disks are block devices. The controller can only read/write an entire sector.
- The "unit of failure" is typically the mechanism.

Redundant Arrays of Inexpensive Disks (RAIDs)

- This chapter provides much detail about aggregating disk mechanisms. We'll summarize briefly.
- There are several RAID configurations, depending on your goals.
- A RAID can increase read performance, since the the first disk to read the data determines the overall time.
- A RAID can improve reliability by storing data on multiple disks, since the "unit of failure" is typically a whole disk.
- Also, since SCSI and SATA disks are "hot swappable" (unless the controller doesn't support it), a failed disk that was "mirrored" can be replaced while the computer keeps running.

39 Interlude: Files and Directories

- We have discussed virtualizing the CPU and memory.
- This chapter describes abstractions above disk drives and their ilk, as well as their drivers, which virtualize persistent storage.

Files And Directories (1 of 2)

- Rather than fixed-size sectors and blocks, people want to manipulate named, variable-length, *files*.
- When storage-device capacities were small, a set of files was sufficient. Now, we want to organize them into a tree of *directories* (aka, *folders*).
- A file doesn't really have a name. It has a unique number (viz., its *inode number*).
- A directory is a file that simply maps a character string (i.e., a *name*) to an inode number. This is important to remember!

Files And Directories (2 of 2)

- A name is a string of one or more characters, other than slash and 8-bit zero.
- A *path* is a sequence of names, separated by slashes, possibly identifying a directory or plain (i.e., non-directory) file. If it identifies a directory, it can have a trailing slash.
- The *root* of the tree is identified by the empty string, and is written as a slash.

The File System Interface (1 of 2)

- The abstraction API is implemented by library functions that typically make system calls. We present the first few with examples:

  `pub/ch39/CopyRW/CopyRW.c`
- Notice that there is no library function or system call to copy a file. A system call strives to be sort-of atomic, in that a process can sort-of expect not to be preempted during one. Some *are* interruptable (e.g., **`read`**), where a interrupted process should retry the system call.

The File System Interface (2 of 2)

- There *are* library functions and system calls to remove or rename a file:

  `pub/ch39/Rm/Rm.c`

  `pub/ch39/Mv/Mv.c`
- You can move a directory with `rename`, but you cannot remove one with `unlink`.
- There is also a function named `lseek`, which changes the “next-byte pointer” of its file-descriptor argument, to its offset argument. This allows overwriting part of, or appending to, an existing file.

Shared File-Table Entries: `fork` and `dup` (1 of 3)

- For each process, the kernel maintains a set of its open-file integer *file descriptors*.
- A file descriptor is a reference to a system-wide *file description*, containing information about a file, including: an offset within the file, and status flags.
- A library call like `open` adds a new file description, and returns a new file descriptor referencing it.
- Library calls like `fork` and `dup` create new file descriptors, which refer to previously existing file descriptions.
- Overall, this allows:
  - a process to have a single file open twice, at the same time, but at different offsets within the file
  - two processes to have a file open, at the same time, at the same offset within the file

Shared File-Table Entries: `fork` and `dup` (2 of 3)

- After a `fork` (`man` page):

  The child inherits copies of the parent's set of open-file descriptors. Each file descriptor in the child refers to the same open-file description as the corresponding file descriptor in the parent. This means that the two file descriptors share open-file status flags, file offset, and ...

Shared File-Table Entries: `fork` and `dup` (3 of 3)

- Sometimes, that's not what you want.
- Instead, you might want the child process to have its own file description. Perhaps, so the child can setup I/O redirection, without interferring with the parent.
- For example, a shell might `fork` a child process, `close` the child's `stdin`, `open` a disk file for it instead, and then `exec` some other program:

  `pub/ch39/Dup/Dup.c`

Memory-mapped Files

- Business-oriented OSs have provided structured files, and even access methods.
- The Unix philosophy is that files should be line-oriented character streams, when possible. This, for example, unleashes the flexibility of pipes.
- However, if your data conforms to a rigid structure, `mmap` can be useful.
- It allocates memory to the calling *process*, by creating a new mapping in the virtual address space of that process.
- Optionally, it is initialized, with bytes from an offset in a file, from a file descriptor.
- Thus, a contiguous data structure can be allocated, and populated with all or some of a file, at once:

  `pub/ch39/Mmap/Mmap.c`

Getting Information About Files

- Directories that map names to a particular file, hold the names of that file. Each file also has other information.
- A file's *metadata* isn't stored in the file, but in its inode. The `man` page for `inode(7)` provides an excellent description.
- A file's metadata can be obtained by the `stat` command, or `stat` library function:

  `pub/ch39/stat.h`
- The `st_mode` field is especially interesting (see: `ls -l` and `chmod`). From low to high:
  - 9 bits are file permissions: 3 bits each (read, write, execute) for user, group, and other.
  - 3 bits are setuid, setgid, and sticky.
  - 4 bits are the file type (e.g., directory).

Making and Deleting Directories (1 of 2)

- The API for directories is much different than that for plain files:

  `pub/ch39/Dirs/Dirs.c`
- A directory can be created (ala, `mkdir`), and an empty one can be removed (ala, `rmdir`). An entry to an existing file can be added (ala, `ln`), or removed (ala, `rm`).
- When no directory, or process, holds a link/reference to a file, *it* can be removed. The `st_nlink` member of the file's inode supports this reference counting.

Making and Deleting Directories (2 of 2)

- The `mkdir` system call creates a link in the current directory to a new, “empty,” directory. Adding a link to a directory is privileged, to avoid cycles, which would pose a security risk. How so?
- However, a *symbolic* (aka, soft) link is often a satisfactory alternative, since tools (e.g., `rm -r`) can discern one and avoid cycles.

Reading Directories

- Accessing a directory is also much different than accessing plain files.
- Here's a larval version of the all-powerful `find` command:

    `pub/ch39/Find/Find.c`

Making and Mounting a File System (1 of 2)

- A bootloader, like GRUB, knows enough about filesystems to locate two files: a kernel and an (initial ram-disk) root filesystem (i.e., an `initrd`).
- GRUB reads the kernel into memory, aims it at the `initrd`, and jumps to its entry point. The kernel reads the `initrd` into memory, *mounts* it at /, and continues. Later, the kernel remounts a disk-backed filesystem at /.
- A process with UID 0, or other appropriate privilege, can make, mount, and/or unmount filesystems, typically via commands, rather than calls to library functions (see the `man` page for `fs(5)`).

Making and Mounting a File System (2 of 2)

- A `mkfs` command writes a new filesystem, of a particular type, to a device (e.g., a disk partition, like `/dev/sda1`).
- The `mount` command/function "attaches" a device, via its filesystem driver, to a directory, called a *mount point*. Traversing a mount point enters the filesystem.
- The `umount` command/function "detaches" a mount point.
- Linux is very flexible in this area. You can: mount a filesystem at multiple mount points, mount a directory at a mount point, mount a filesystem within a mounted filesystem, and so on.

File System Implementation

- We have discussed filesystem abstractions, and how a program uses them. This chapter, and the sequel, explore implementing those abstractions, in more detail than we really need.
- Memory and filesystems are both storage. They differ primarily in access speeds, access patterns (i.e., random versus sequential), and persistence.
- A filesystem's code can be compiled into a kernel, or loaded as a kernel module. Mainly, it is a driver that provides an abstraction, atop a disk driver.
- A disk driver abstracts physical sectors (e.g., of 512B) into logical sectors (e.g., again, of 512B), which a filesystem then abstracts into blocks (e.g., of 4KB). See `tune2fs` and `lsblk`. Also:

  `pub/ch40/blocksize`

The Way to Think

- In addition to its code, a filesystem comprises in-memory and on-disk data structures.
- The in-memory part is mainly cache buffers, for performance.
- The on-disk part supports allocation, deallocation, free-space management, and defragmentation. All of this is at block-size granularity.
- Inodes, directories, and plain files must be accommodated. An inode is mainly an on-disk `stat` structure.

Locality and The Fast File System (1 of 2)

- Ken Thompson wrote the first Unix filesystem, in the early 1970s. He's now at Google, where he helped develop the Go language. He has won a Turing Award.
- This filesystem treated the disk like random-access memory. Data was spread-out all over it. It also suffered from fragmentation.
- Data-transfer bandwidth could drop to 2% of that provided by the raw disk.

Locality and The Fast File System (2 of 2)

- Without going into gory details, the basic improvement of the subsequent "Fast File System" was to store data that needed to be accessed sequentially on nearby tracks/cylinders, preferring rotational delays to seek delays.
- The idea was: *keep related stuff together* and *keep unrelated stuff far apart*.
- Fragmentation was reduced by having two block sizes. In addition to 4KB blocks, "sub-blocks" of 512B supported small files and avoided mainly empty blocks.
- Other features included longer filenames (i.e., more than eight characters) and symbolic links.

Crash Consistency: `fsck` and Journaling (1 of 2)

- Filesystem corruption is a scary prospect.
- The on-disk data structures that advantage locality and reduce fragmentation are complex, and tricky to reconstruct.
- There are two main causes of inconsistency:
  - a sequence of disk operations that needs to be atomic, but is interrupted
  - a cached disk operation that needs to be committed to disk, but is interrupted
- Since delaying/preventing infrequent interruptions (e.g., power outages) is very expensive, subsequent recovery is preferred.

Crash Consistency: `fsck` and Journaling (2 of 2)

- There are two main recovery approaches.
- Prior to mounting a filesystem, scan the entire filesystem (perhaps with multiple passes), look for various kinds of inconsistency, and fix them. This is what `fsck` does. For a large filesystem, this can take hours.
- Prior to writing to a filesystem, make a small "log entry" to a well-know area within the filesystem. Prior to mounting a filesystem, check the log for any uncompleted operations, and replay them. This is called *journaling*.
- An `fsck` speedup is to mark a properly unmounted filesystem as "clean," and only perform checking for a filesystem that was unmounted "uncleanly" (or hasn't been checked for a while).

Other Approaches

- Our authors give much detail (this is their research area), but we should consider the *copy-on-write* (COW) approach.
- It can be applied to memory and storage, to save space. Subversion does this.
- Here, we only care about COW storage, for fault tolerance (e.g., a filesystem).
- When a file is copied, don't really make a copy, just have two references to the same file.
- When a file is modified, really make a copy, changing the copy as needed (even if only one byte changes).
- After an interruption, damage is limited to only the last operations, which are easily undone. A journal allows replay, too.