// solutions to practice exercises:
// http// solutions to practice exercises:
// http://os-book.com/OS9/practice-exer-dir/
// whenever possible C programming assignment will be done in Rust

1.1
as stated in the text,
"In general, we have no completely adequate definition of an operating system."
more or less an operating system has to
1. provide an interface of hardwares to softwares (may not be the case for embedded systems)
2. run the user applications in a safe manner (again may not be the case for embedded systems)
3. manage hardware resources

1.2
1. when strictly uniform performance is of importance
e.g. the operating system has to perform exactly the same on different hardwares, maybe for real-time purpose
2. for security reasons
e.g. the recent Intel CPU exploits which operating systems sacrifices ~15% performance to patch
3. to provide a backward-compatible user interface to softwares
the operating system have to support legacy softwares and run in a mode designed for hardwares decades ago
all these "waste" are necessary to fulfill their design requirements

1.3
the running time of all the instructions must be deterministic, which means:
GC is almost impossible: the operating system cannot randomly pause for a milliseconds and collect garbages
the language in use must have a very strict semantic model, mapping program to instructions in a well-defined manner
some probabilistic algorithms and data structures cannot be utilized, e.g. hashtable

1.4
should:
web browser is an absolutely necessary utility which should be provided by an operating system out-of-box
compiling an entire web browser is both painful and difficult, web browsers are usually distributed in binary
shouldn't:
it isn't really part of an operating system, web browser definitely doesn't run in kernel mode
for linux distros most of their users should know how to fetch a web browser with wget or curl

1.5
user applications cannot run kernel mode privileged instructions directly which may potentially interfere the operating
system and other user applications
they may only delegate access to these instructions to the operating system by means of well-defined syscalls
so these syscalls can be analyzed and verified by the operating system in the context of processes

1.6
a. privileged, otherwise a user application can hold the CPU forever
b. not privileged
c. privileged, otherwise user applications can interfere with each other unchecked
d. not privileged, by design
e. privileged, otherwise a user application can deny syscalls from other applications and hold the CPU forever
f. privileged, hardware is managed by the operating system, not directly by the user applications
g. privileged, otherwise renders the whole dual-mode system useless
h. privileged, IO devices are managed by the operating system, user applications should use syscalls

1.7
1. the operating system, loaded into memory by the firmware on startup from secondary storage, cannot be modified by
   the operating system itself, i.e. no update
2. the protection is not complete: an operating system is not entirely static, it has to hold dynamic memories
   for e.g. file system

1.8
ring -3: Intel ME, a separated chip beside Intel CPU which control and monitor the entire computer from beneath
ring -2: processor microcode, under machine code instructions, the real instructions running on the chip
ring -1: hypervisor of the virtual machine environment, manage multiple virtualized operating systems
ring 0: kernel, operating system

1.9
starting from a time set by the user or synced from a remote server:
   the process asks the kernel to wake it a fixed time later by a syscall and go to sleep
   the operating system schedules the process to run after the fixed time

the process upon wake up and increment the time by the fixed time

1.10
cache is useful due to the fact that:
they are faster in term of access time compared to the storage on levels below
by caching multiple small read / write can be performed in batch, possibly extending the lifetime of the storage
if it can only survive a finite time of read / write operations
cons: caching complicates access to the storage, also cache invalidation is one of the only two hard things in compsci
even if the cache is as big as the storage device, it may not be persistent, the content still have to be written to the
storage device before power off

1.11
in client-server distributed systems hosts either generate or satisfy requests, in peer-to-peer distributed systems
every host do both of them

1.12
a. 1. without proper isolation one user may have access to all the sensitive contents belongs to another user
2. critical failure of one user process may affect or even terminate other processes
b. virtualization may be able to provide the security, but in practice there's always another exploit yet to be found

1.13
a. CPU, memory, storage devices
b. CPU, memory, IO devices, network links
c. (in addition to workstations) battery

1.14
1. when the user only use no more than a fraction of the hardware capacity, part of a time-sharing system can be
rented instead of an entire PC or workstation
2. when the user works in a group and have to share read / write access to some files within the group, it's cheaper to
work on a time-sharing system then doing so via network links

1.15
symmetric:
all CPU cores are equal in functionality, with the same amount of SRAM cache, computing power and access to the
main memory, tasks are distributed in software level as well as communication between cores

common design of desktop CPUs
asymmetric:
CPUs have different functionality: a main CPU controls how work is distributed among worker CPUs
software software communicates solely to the main CPU
common design of mobile CPUs
pros (compared to multiple single-processor PC):
cheaper, no duplicated IO device and IO bus
communication / memory sharing among processors are faster by magnitudes
easier to program: just spawn threads and the operating system can distribute them evenly among processors
cons:
it is a single PC with a single set of IO device at the end, cannot be used by two people at the same time

1.16
instead of multiple processors managed by a single operating system, clustered systems "are composed of two or more
individual systems—or nodes—joined together"
hardware: a network, wired or wireless
software: a distributed operating system, running on hosts and monitors each other

1.17
1. duplicate each write to the two nodes, distribute reads evenly, higher availability, half efficiency
2. distribute write evenly between two nodes, properly guide reads to the correct node (e.g. by DHT)
full efficiency, a hardware failure permanently wipes data

1.18
a network computer (thin client) passes most of its tasks, which would be computed locally by a traditional PC, to a
remote server assigned to it
by handing over its tasks, network computer consumes less energy thus has longer battery lifetime
if the data is stored in the server as well, users may access their data from different terminals seamlessly

1.19
to inform the CPU about a software / hardware event so it can handle it
traps are software generated interrupts, indicating either an software error (e.g. divide by zero) or a request of
operating system service (syscalls)

1.20
a. CPU sets up buffers, pointers and counters for the DMA controller, which copies a whole block of data into the
memory accordingly, meanwhile the CPU can do something else
b. by a hardware interrupt
c. thanks https://en.wikipedia.org/wiki/Direct_memory_access#Cache_coherency
    the memory transfer is invisible to the CPU before completion, DMA may cause cache incoherency between CPU cache and
    the main memory

1.21
the published instruction sets may be an abstraction over the real instructions running on the hardware (microcode),
hence a dual-mode instruction set can be build on a mode-less hardware
such an architecure may be exploitable to adversaries with physical access to the computer

1.22
to better handle the typical computing tasks: in practice user applications works on both shared and isolated memories

1.23
depends on the memory ordering of read / write accesses, a write operation performed by one processors may or may not
be visible to other processors immediately
refer https://doc.rust-lang.org/stable/std/sync/atomic/enum.Ordering.html

1.24
a. DMA may perform memory write temporarily invisible to CPU and cache
b. different cores may perform independent read / write thus have different cache before synchronization
c. same to multiprocessor

1.25
dual-mode instructions
memory access is privileged instruction, user applications can only perform memory access through syscalls, operating
system maintains memory region of each process, upon receiving a syscall examines if the process is accessing its own
memory or not, if not return the the infamous segmentation fault error to the user application

1.26
a. LAN, all devices are physically reachable by a network administrator
b. WAN
c. WAN, a neighborhood in general is not an organization

1.27
limited power supply
limited computing power
different IO devices (touch screen instead of keyboard & mouse)
higher requirement of stability (phones are rarely turned off)

1.28
higher availability: client-server architecure has a single point of failure which is the server
better scalability: peer-to-peer architecure in nature is more scalable compared to extending the capacity of a server

1.29
file distribution (e.g. bittorrent)
video streaming
network traffic tunneling (e.g. TOR)

1.30
pros:
  more participants, earlier bug detection, more agile response to needs (for developers)
  lower cost (for consumers)
  provides learning material to students
cons:
  bigger attack surface
  copies cannot be sold

2.1
provide an compatible and easy-to-use interface to user softwares and higher layer system components
hide the real implementation of system functionalities which is subject to change

2.2
// section 1.6
create and deletion
suspension and resumption
synchronization
communication
deadlock handling

2.3
// section 1.7
keeping track of which parts of memory are currently being used and who is using them
deciding which processes (or parts of processes) and data to move into and out of memory
allocating and deallocating memory space as needed

2.4
// section 1.8
free space management
storage allocation
disk scheduling

2.5
to provide a programmable, concise and unambiguous user interface to run system and user applications
it's not implemented as a module of the kernel because it doesn't have to, a shell can be built on top of syscalls

2.6
fork() to create a process inheriting the stack of the current one

2.7
enhance modularity of the operating system, part of the operating system can be built upon the kernel instead of in it

2.8
modularity, the operating system can be implemented, reasoned and debugged one part of a time
but layered approach increases the overhead of syscalls

2.9
GUI
  human friendly access to user and system programs
  a user program in Linux land but a system program in Windows and OSX
  in a closed-source OS, the GUI usually is hard-coded into the system
program execution
  OS essential
  user program cannot access the layer under syscall
IO operation
  OS essential
  user program cannot access the layer under syscall
file system manipulation
  organize file in a human readable way
  it must be part of the OS, user program cannot define their own flavor of file system on the one provided by the OS
resource allocation
  OS essential

    kind of doable in user land if the OS exposes lower level syscalls to memory management functions
    earlier version of Rust used a custom memory allocator instead of the OS default

2.10
in that way the OS doesn't have to be loaded into the main memory
those hardwares simply have no enough memory to load the system

2.11
operating systems write their location on the disk to a special boot block on the disk during installation
on startup the boot loader reads these locations then let the user choose one from them

2.12
1. UI and exposed API: load, IO, file system, communication, error detection
2. internal system functions: resource allocation, resource accounting, memory protection

functions in the first category is visible to the user in form of exposed API and syscalls
functions in the second category is not visible but a promise the OS makes to the user

2.13
1. put in a predefined sequence of registers, e.g. first few parameters in Linux ABI
2. pushed to stack, e.g. after a few parameters in Linux ABI
3. stored in a table in the memory

2.14
build the program with debug symbols, let the operating system interrupt the program frequently with a timer, record
where the execution is each time the program is interrupted
without time profiling optimization of a program of decent complexity is nearly impossible

2.15
// section 1.8.1
creating and deleting files
creating and deleting directories
file and directory manipulating primitives
mapping files onto secondary storage (mmap?)
backup to stable storage media

2.16
pros:
    these two concepts share a lot of resemblance
    a common API to both of them simplifies the system design and its interface
cons:
    not all file operations are possible to all devices

e.g. seek a tty device results in an error
certain devices are too danger to be manipulated as a file, pipe something to /dev/hda will wipe the boot block

2.17
possible if the default one is implemented in user land, i.e. all it does is call system programs

2.18
message passing: safer with bigger overhead and requires extra memory
shared memory: faster but must be done right carefully or will cause data race / deadlock

2.19
like any other program, the resulting operating system will be more configurable

2.20
device driver and memory management
memory management must be built upon drivers, but drivers may also need access to the memory

2.21
all but the bare minimum functions of the operating system is implemented as separate programs
user program requests services from the kernel
which in turn passes these requests to the corresponding module program
kernel components in microkernel architecure communicates by inter-process communication, much slower than function
calls in the same program

2.22
so these functions can be implemented and compiled separately and dynamically
installation of a new device can be done without recompiling or even restarting the system

2.23
they are both mobile operating system, serving roughly the same set of hardware and market
iOS is closed-source, Android is partially open-source

2.24
https://web.archive.org/web/20080604070631/http://blogs.sun.com/jrose/entry/with_android_and_dalvik_at
main reason listed: GPL of openJDK, power consumption of JVM, efficiency of assembly v.s. Java bytecode

2.25

https://www.semanticscholar.org/paper/An-Overview-of-the-Synthesis-Operating-System-Pu-Massalin/c8f49d5d94e1f9adf608b7c99e1e731c2381d82a
Factoring Invariants method
    if a system service is called frequently with the same parameters, the execution path of the call is cached and
    carried out next time the same service is called
Collapsing Layers method
    lower layer of services can be invoked directly in execution
Executable Data Structures method
    traversal order is stored alongside the data in data structures
faster syscalls in exchange of less flexible operating system architecure

2.26

```
./OS/posix-file-copy
sudo dtrace -n 'syscall:::entry /execname == "posix-file-copy"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 535 probes
^C
  __mac_syscall                                                     1
  access                                                            1
  bsdthread_register                                                1
  exit                                                              1
  fcntl                                                             1
  getrlimit                                                         1
  issetugid                                                         1
  shared_region_check_np                                            1
  sysctlbyname                                                      1
  thread_selfid                                                     1
  close                                                             2
  csops_audittoken                                                  2
  fcntl_nocancel                                                    2
  fsgetpath                                                         2
  fstat64                                                           2
  getentropy                                                        2
  getpid                                                            2
  proc_info                                                         2
  read_nocancel                                                     2
  write                                                             2
  ioctl                                                             3
  read                                                              3
  csrctl                                                            4
  open                                                              4
  mprotect                                                         10
  stat64                                                           42
```

3.1
PARENT: value = 5
child process inherits a copy of stack of the parent process, modification in the child process will not reflect to the
parent process

3.2
8

3.3
context switching
short-term process scheduling
accounting of process memory and inter-process communication

3.4
a special syscall switches the CPU to the other register set
some register set has to be swapped out to the memory to make room for the new context

3.5
only c. shared memory segments, others are copied

3.6
similar to the design of TCP
=> request
<= process call
    ACK (lost)
=> request (resend)
<= duplicate sequence number, ignored
    ACK
=> ACK received

3.7
retransmission

3.8
short-term scheduling:
    controls which process in the ready queue could run and allocates CPU time to it
    executes once every ~100ms
medium-term scheduling:
    controls which process should be kept in memory
    swaps a process out from memory is it's waiting on an IO event or others
    executes on every major IO event
long-term scheduling:
    controls in which order pooled processes could run

    executes every time a process terminated

3.9
saves the context of the current process to its PCB structure
loads the context of another process from its PCB structure

3.10
// from a fresh WSL debian
init(1) -> init(2)
init(2) -> bash
bash -> ps

3.11
when a process is terminated without waiting on its child processes, its child processes become orphans
the OS assigns init as the new parent of these orphan processes
init calls wait periodically so these orphan processes can be terminated on completion

3.12
16

3.13
in the child process if the call to fork succeeded

3.14
A: 0
B: 2603
C: 2603
D: 2600

3.15
ordinary pipe:
    message passing between parent and child processes
named pipe:
    abstract over the functionality of some program to provide an standard file interface to other programs
    e.g. a pipe that compresses the data passed to it and store them in a file

```
// https://en.wikipedia.org/wiki/Named_pipe
mkfifo my_pipe
gzip -9 -c < my_pipe > out.gz &
```

3.16
at most once:
    an RPC service on an embedded system controlling a light bulb

    the bulb blinks instead of being switched on / off
exactly once:
    a distributed file system
    a write operation may be lost due to connection failure
any use case of UDP is also a use case of unreliable RPC

3.17
CHILD:
    0, -1, -4, -9, -16
PARENT:
    0, 1, 2, 3, 4

3.18
a. synchronous communication:
        if blocks on both ends the messages are passed from producer to consumer directly, no buffer required
        same as other blocking operations wastes cycles
    asynchronous communication:
        easier to use for programmers
        the messages have to be buffered somewhere
b. either the programmer or the kernel has to carry the burden of buffering
c. extra memory copy & occupation v.s. danger of dangling pointers & data race
    OS must be able to mark memory as shared in the latter case
d. fixed-sized messages: easier to implement but harder to use
    variable-sized messages: the opposite

3.19
./OS/process/zombie-child

```
% ps -l
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY
TIME CMD
  501  2037  1916     4006   0  31  0  4297288   2136 -      Ss                0 ttys001    0:00.03
/bin/zsh -l
  501  2279  2037     4006   0  31  5  4268544    740 -      SN                0 ttys001    0:00.05
target/debug/zombie-child
  501  2289  2279     2006   0   0  5        0      0 -      ZN                0 ttys001    0:00.00
(zombie-child)
```

3.20
// skipped, would be a simple wrapper around a hash set in this stage

3.21
./OS/process/collatz-child

3.22
./OS/process/collatz-child-shm
apparently there's no way to list and remove shared memories in OSX which one forget to unlink

3.23 - 3.25
// skipped, entry level application layer programming

3.26
./OS/process/case-reverse-pipe

3.27
./OS/process/file-copy-pipe

Project 1 — UNIX Shell and History Feature
./OS/process/unix-shell

4.1
1. a web server, single-threaded server can only handle one client at a time even though each client only need a tiny
   amount of computation power and is waiting for IO most of the time
2. video compression on a multi-processor computer, a single-threaded program can only ultilize one of many cores,
   even middle-range CPUs today has dozens of cores

4.2
user threads are
   1. directly created by user program or a thread library
   2. managed solely by user space code
where kernel threads are
   1. created by the operating system in a pre-defined model (M:M, M:N or M:1) to meet the request of user threads
   2. managed by kernel space code

4.3
store and load stack, register and thread local storage
in contrast to process context switch, file system, VM space, file descriptors and signal handlers are not switched

4.4
same as above, in general fewer memory is allocated for threads than for processes

4.5

it must, otherwise one thread blocking on IO will block the other threads on the same LWP as well
this kind of blocking and pause is unpredictable and unacceptable in real-time system

4.6
1. when there's only one processor, any CPU bound program will not have better performance with multithreading
2. when the program is IO bound but they all access the same IO device and the IO device can only serve one thread at
   a time in a very slow pace, multithreading will not provide any significantly better performance

4.7
when they are IO bound accessing different IO devices, instead of the sum of the IO response time, only the maximum of
the IO response time would be necessary

4.8
b and c

4.9
when they are CPU bound, on a multi-processor system all cores can be utilized

4.10
if the program is carefully designed so that the error in client script won't corrupt memory shared among threads
the isolation may still be achieved if the signals can be handled locally to the thread

4.11
as stated in the text: multithreaded program in a single processor system

4.12
a. 10 / 7
b. 20 / 11

4.13
a. only task, the same data set is shared
b. both task and data
c. both task and data
d. task but not data, the same data (web pages) are accessed by each thread

4.14
a. one, this task won't benefit from multithreading
b. depends, if the tasks are uniform 4 is enough, otherwise finer granularity is necessary or some CPU would be idle

4.15
a. 6
b. 2, or 4 if threads are duplicated by fork()

4.16
as stated in the text, the former manages data associated to a task (process or thread) separately , storing only
pointers to the data in the PCB, while the former has different data structure for processes and threads

4.17
CHILD: value = 5, global variables is shared (unsafely) among threads
PARENT: value = 0, each process has its own copy of global variables

4.18
a. the full computing power of the system will not be made use of
b and c may not be distinguishable from the view of the program

4.19
1. resource release e.g. closing a file descriptor
2. write to a log file

// about the difference between Pthread API and Rust thread API:
// they are semantically equivalent but:
// 1. threads in Pthread take input by a void pointer on thread spawning, while Rust threads are a closures
// 2. Rust thread can return a Sized value, which can be emulated with Pthread API by allocate a dedicated chunk of
//   memory for the thread to store the return value and later accessed by the main thread
4.20
./OS/process/pid-manager
a lock (mutex) is necessary to prevent data race
not sure if such a lock or variable length vector is available in kernel space

4.21
./OS/process/multi-thread-statistic

4.22
./OS/process/monte-carlo-pi

4.23
// skipped

4.24
./OS/process/prime-numbers

4.25
// skipped, similar exercise in CSAPP

4.26
./OS/process/fibonacci

4.27
// skipped, similar exercise in CSAPP

Project 1

Project 2
// skipped, similar exercise in CLRS

5.1
system clock is implemented by periodical clock interrupts
if interruption is disabled system-wise, the clock interrupts may be delayed, the assumption that these interruption
occurs once a fixed time period no longer holds
either interruptions of the clock thread should not be disabled for too long, or the system time should be synchronized
to a remote server from time to time

5.2
these synchronization primitives have different semantics and diffferent costs
spin lock:
  busily wait by testing and modifying a shared variable atomically
  used when the critical section is short, i.e. syscalls and interrupts are significant overheads in comparison
mutex lock:
  semantically equivalence of spin lock provided by the kernel
  instead of atomic instructions in a busy loop, the threads waiting for the lock is put to sleep and awaken later
  more efficient than spin lock if the critical section is long
semaphore:
  similar to mutex lock, but instead of a binary lock (one thread working, all other threads waiting), (a fixed
  number of) multiple threads can enter the critical section at the same time
  used when a fixed number of resources have to be managed across threads
adaptive mutex lock:
  a hybrid implementation of lock as both a spin lock and a mutex lock

when the lock is held by a working thread, threads blocking on waiting spin
when the lock is held by a sleeping thread, threads blocking on waiting is put to sleep too

condition variable:
a synchronization mechanism defined by two methods: .wait() and .signal()
x.wait() blocks the current thread until another thread calls x.signal()
x.signal() awakes a thread or all threads blocking on x.wait()

5.3
waiting by busily trying to acquire a lock in a loop
alternatively a thread can be put to sleep (by a syscall or a kernel mutex lock) and be awaken later by the OS when the
resources become available
it's not always efficient to avoid busy waiting, these syscalls and interrupts have their own costs

5.4
usually the kernel cannot tell the difference between a thread doing meaningful job and a spinning thread
on single-processor system the spinning thread when active will consume 100% of the computing power of the CPU
also for a resource to be released, the spinning thread must be switched out for another thread to make progress
on multi-processor system the spinning thread will only consume 1/N of the computing power where N is the number of
processors in the system
the resource may be released by another thread running on another processor, switching out the spinning thread may be
much more expansive compared to spinning a few hundred instructions

5.5
without atomic operations the memory order may be arbitrary, decrementation of the counter in one thread may not be
visible from another thread in time, two or more threads may acquire the resource guarded by a binary semaphore

5.6
a binary semaphore is semantically equivalent to a mutex lock

5.7
assume the instructions are interleaved in order:

```
LD R0 BAL   LD R1 ACC
ADD R0 A0
        SUB R1 A1
ST R0 BAL
        ST R0 BAL
```

the BAL memory location storing the account balance may have value BAL - A1 or BAL + A0 while it should have value
BAL + A0 - A1
the operations to the same account must be guarded by a mutex lock

5.8
assume turn is initialized to either i or j (otherwise the program causes UB)
assume all operations on variable turn and flag[i] are atomic
mutual exclusion:
    Pi will only enter its critical section when flag[i] && !flag[j] and vice versa
    the two conditions cannot be satisfied at the same time
progress:
    if Pj is absent when Pi is trying to enter the critical section, flag[i] && !flag[j]
    otherwise flag[i] && flag[j]
    if turn == i Pi spins, otherwise Pi set flag[i] = false hence Pj can make progress and set turn = i on exit
bounded waiting:
    when two threads compete for the resource, only the thread pointed by variable turn can make progress
    on exit turn is set to another thread, so a blocking thread at most have to wait for another thread to exit

5.9
// thanks
https://personal.cis.strath.ac.uk/sotirios.terzis/classes/CS.304/Remaining%20Contemplation%20Questions.pdf
mutual exclusion:
    the first thread Pi entering critical section has checked that
    1. itself has state in_cs
    2. all other threads has state != in_cs
    any thread entering state in_cs later cannot exit the first while loop
progress:
    if multiple threads are in state want_in, all but the one immediately after turn is stuck in a spin lock
bounded waiting:
    on exit, threads increments turn by at least 1 (skips idle threads)
    a waiting thread cannot be skipped, after at most n - 1 turns this thread will be served

5.10
in single-processor system it's important that a single thread should not hold the CPU too long, which is enforced by
a timer, otherwise a dead lock will freeze the entire system
disabling all interrupts will disable the timer interrupt at well, let a single thread hold CPU indefinitely

5.11
disabling / enabling interrupts to all threads on demand of a single thread is way too expensive

5.12
// thanks https://stackoverflow.com/questions/4752031/
trying to acquire a semaphore may put the thread to sleep, which defeats the purpose of spin lock: spin locks should
only be used if the critical section is short, sleeping with a spin lock will block all other threads trying to acquire
the spin lock for a long time

5.13
linked list:
  if a thread is deleting a node when another thread is trying to insert a node before it, the pointer to the previous
  mode may be dangling, the inserting thread would be dereferencing NULL pointers and cause segfault
bitmap:
  setting the same field to both 0 and 1 in different threads may have varying result depends on the order of their
  instructions

5.14
assign id 1 to n to n threads
each thread acquire the lock by
```
   while (compare_and_swap(&lock, i, 0))
```
and set the lock to the next non-idle thread based on a flag[n] array

5.15
```
void acquire(lock *mutex) {
   compare_and_swap(&lock->available, 0, 1);
}
void release(lock *mutex) {
   test_and_set(&lock->available, 0);
}
```

5.16
instead of while loop, acquire invokes a syscall with the lock (implemented in the kernel), the kernel then put the
thread to sleep and insert it into a waiting list associated to that lock

5.17
short duration: spin lock

long duration: mutex or spin lock, depends on the number of cores available and the time
sleep while lock: mutex by 5.12

5.18
2T

5.19
the second one is much more efficient, the variable hit is never read, its value is of no importance to the service
threads, a mutex lock may put threads to sleep while atomic operations will be compiled to a hardware instruction

5.20
a. all operations on number_of_processes may cause data race, the real number of processes exceed MAX_PROCESSES
b. before each read and write to number_of_process
c. without a mutex lock two threads may call allocate_process() at the same time when
    number_of_processes == MAX_PROCESSES - 1
   the maximum number of processes will be exceeded

5.21
initialize a semaphore to the maximum number of sockets
a thread opening a connection has to acquire the semaphore at first, block if the semaphore is 0

5.22
such a lock may be useful when concurrent quick read is frequent, write is sparse
if write is as frequent as read or the read operating is long, the threads trying to write may be starving

5.23
assume read operation to semaphore is atomic

```
typedef struct {
    boolean block;
    int count;
} semaphore;
void wait(semaphore *lock) {
    while(true) {
        while (test_and_set(&lock->block))
            ;
        // acquired lock
        if (lock->count > 0) {
            lock->count -= 1;
            break;
        }
    }
}
void signal(semaphore *lock) {
    while (test_and_set(&lock->block))
        ;
    lock->count += 1;
}
```

5.24
the child thread must inform the parent thread about a ready number in some means (e.g. condvar), or while very
inefficient the parent thread can check an atomic count of ready numbers in a loop
./OS/process/fibonacci-iterative

5.25
mutual exclusion of a monitor can be emulated by a semaphore guarding all methods of that monitor

5.26
// skipped, straight forward

5.27
// thanks solutions manual
a. if the content is lengthy, copying it to / from the buffer may take considerable time, another thread trying to
    read or write the buffer must wait in between
b. instead of produced buffer elements, the buffer may store pointers to these elements, copying these pointers would
    be far more efficient
    threads must be careful that these memory region should not be accessed once its put into the shared buffer

5.28
in a common use case of read-write lock, there will be much more read threads than write threads
if the lock does not favor write threads, a write attempt will block until there's no thread reading the resource
such an event may simply never happen if the resource is read frequently
a possible solution is, if there are write threads waiting on the lock, all incoming read threads are blocked
in this case write threads are will starve but read thread may

5.29

calling signal() on a conditional variable awakes one thread blocking on the corresponding wait() call, it there's no
threads blocking on the conditional variable the call to signal() has no effect
in contrast calling signal() on a semaphore will always release one allowance to the poll

5.30
threads calling signal() no longer has to wait the awakened thread to exit
all operations on the next semaphore and next_count variable can be eliminated

5.31
along with the resources and conditional variable, the monitor maintains a waiting priority queue
when no printer is available, incoming threads enter the waiting queue and wait on the conditional variable
whenever a printer is released, all threads waiting on the conditional variable are awaken (pthread_cond_broadcast)
they in turn check if they are the top of the priority queue and there's printer available, if so they acquire a
printer, otherwise they call wait() again

5.32
with Rust Condvar and Mutex:
```
struct FileSharing {
    sum_lock: Mutex<u32>,
    cv: Condvar,
    limit: u32,
}
impl FileSharing {
    fn new(limit: u32) -> Arc<Self> {
        Arc::new(Self {
            sum_lock: Mutex::new(0),
            cv: Condvar::new(),
            limit,
        })
    }
    fn acquire(&self, num: u32) {
        let mut sum = self.sum_lock.lock().unwrap();
        while *sum + num >= self.limit {
            sum = self.cv.wait(sum).unwrap();
        }
        *sum += num;
        // acquired file, enter critical section somewhere else
    }
    fn release(&self, num: u32) {
        let mut sum = self.sum_lock.lock().unwrap();
        *sum -= num;
        drop(sum);
        self.cv.notify_all();
    }
}
```

5.33
upon release the calling thread is no longer holding shared resources, so it makes more sense to continue execution in
this case, if the control is transferred to the notified thread the calling thread may have to call release() in
async manner, e.g. spawn another thread to call it

5.34
a.

```
monitor rw_lock {
        int read_count;
        int write_count;
        initialize() {
            read_count = 0;
            write_count = 0;
        }
        read() {
            await(write_count == 0);
            read_count += 1;
            // critical section
            read_count -= 1;
        }
        write() {
            await(read_count == 0 && write_count == 0);
            write_count += 1;
            // critical section
            write_count -= 1;
        }
    }
```

b. there's no mechanism there to notify the threads and the await() function that the value of the expression has
changed, the expression has to be evaluated from time to time even if the expression is expensive
also the system instead of the user program has to decide which thread to awake first when the condition is
satisfied, without specific knowledge of the user program the decision made by the system may not be efficient

c. the paper in mention is paywalled

maybe the expression has to be simple boolean expressions of local variables so its value can be tracked

5.35
with Rust Condvar and Mutex:
```
struct AlarmClock {
    clock_lock: Mutex<u32>,
    cv: Condvar,
}
impl AlarmClock {
    fn new() -> Arc<Self> {
        Arc::new(Self {
            clock_lock: Mutex::new(0),
            cv: Condvar::new(),
        })
    }
    fn wait(&self, ticks: u32) {
        let mut clock = self.clock_lock.lock().unwrap();
        let now = *clock;
        while clock.wrapping_sub(now) < ticks {
            clock = self.cv.wait(clock).unwrap();
        }
    }
    fn tick(&self) {
        let mut clock = self.clock_lock.lock().unwrap();
        *clock = clock.wrapping_add(1);
        self.cv.notify_all();
    }
}
```

5.36
// already done in 4.20
./OS/process/pid-manager

5.37
a. available_resources
b. two call to either decrease_count() or increase_count() may cause inconsistent write to the variable
c. ./OS/process/resource-counter

5.38
./OS/process/resource-counter

5.39

./OS/process/monte-carlo-multi

5.40
// skipped, no OpenMP in Rust

5.41
./OS/process/barrier-point

Programming Project 1
./OS/process/sleeping-ta

Programming Project 2
./OS/process/dinning-philosophers

Programming Project 3
// skipped, simply a VecDeque in a Mutex

6.1
n! if the system is non-preemptive
(Σti / q)! if the the system is preemptive, length of ith process is ti, the time quantum is q

6.2
non-preemptive scheduling will not pause a process and switch context unless it exits / waits on resources by a syscall
preemptive scheduling may pause a process midway to ensure fair share or to start a process of higher priority

6.3
a. P1: 0 -> 8
   P2: 8 -> 12
   P3: 12 -> 13
   avg turnaround: 10.533
b. P1: 0 -> 8
   P3: 8 -> 9
   P2: 9 -> 13
   avg turnaround: 9.533
c. P3: 1 -> 2
   P2: 2 -> 6
   P1: 6 -> 14
   avg turnaround: 6.866

6.4
processes with high priority are usually interactive, a shorter time quantum will let it be executed more frequently

hence reduce the response latency
processes with low priority are usually CPU bound daemons doing heavy computing job, a longer time quantum will reduce
the time wasted on context switching

6.5
a. SJF is a special form priority scheduling, priority is decided solely by burst time
b. FCFS is a special feedback queue with a single queue running FCFS algorithm
c. FCFS is a special priority scheduling where all processes are assigned equal priority and ties are broken by FCFS
d. different, RR is preemptive and SJF is non-preemptive

6.6
a process with low priority (e.g. a CPU-bound processes) will eventually be assigned higher priority since it's never
executed in the recent past due to its low priority

6.7
on systems with M:1 or M:N thread model, the thread library uses the PCS scheme to map user threads to LWP where threads
in a process compete for LWP, then the OS uses SCS scheme to map LWP to a physical CPU where LWPs system-wide compete
for the hardware
switch to SCS scheme on an M:1 or M:N system effectively turns its thread model to 1:1

6.8
same to 4.5

6.9
P1: 80
P2: 69
P3: 65
lower

6.10
the same to 6.4: IO-bound programs should have lower response time and CPU-bound programs should be preempted less

6.11
a. lower response time can be enforced by shorter time quantum and preemptive system, but these changes cause more
   context switch and lower CPU utilization
b. as illustrated in 6.3, to minimize average turnaround time the OS should wait for a short period when there's only

a few ready processes, which of course increases the average waiting time
c. better IO utilization means more syscalls, more interruption and more context switch

6.12
more than one lottery tickets can be assigned to a process, increasing its winning probability accordingly

6.13
1. pros:
    no shared memory, queues can be safely manipulated by the core at any time
   cons:
    without some other inter-core scheduling mechanism, the processes have to wait in the queue of a busy core even
    if another core is idle, result in worse load balance
2. pros:
    load balance is naturally assured, all idle cores will get processes from the global queue
   cons:
    the queue must be protected by synchronization tools, access to the queue may be unsafe or at least take
    longer time

6.14
a. the prediction is always 100ms
b. the CPU burst history has almost no effect on the prediction

6.15
CPU bound, a process which do not invoke a syscall for IO in its time quantum will have a higher priority and longer
time quantum next time it's executed

6.16
a. FCFS:
    0 -> 2: P1
    2 -> 3: P2
    3 -> 11: P3
    11 -> 15: P4
    15 -> 20: P5
   SJF:
    0 -> 1: P2
    1 -> 3: P1
    3 -> 7: P4
    7 -> 12: P5
    12 -> 20: P3
   priority:

0 -> 8: P3
8 -> 13: P5
13 -> 15: P1
15 -> 19: P4
19 -> 20: P2

RR (time quantum in order):
P1, P2 (t = 1), P3, P4, P5, P3, P4, P5, P3, P5 (t = 1), P3

b. see part a

c. turnaround time - burst time
result skipped

6.17
// skipped, unclear how the priority should affect the RR schedule

6.18
if all the user threads are assigned a 0 nice value, decreasing the nice value of a process without proper permission
may be considered as privilege escalation

6.19
b and d, a if the system is non-preemptive and the first coming process does not halt

6.20
a. the process is executed twice as frequent as other processes
b. alongside the pointers store a time quantum specific to that process

6.21
a. one round consists of: 11 CPU burst of length 1 and 11 context switch
11 + 0.1 * 11 = 12.1 ms in total, CPU utilization is 11 / 12.1 = 0.909
b. one round consists of: 10 CPU burst of length 1, 1 CPU burst of length 10, 11 context switch
10 + 10 + 0.1 * 11 = 21.1 ms in total, CPU utilization is 20 / 21.1 = 0.948

6.22
assign higher on priority to their process on creation, invoke dummy syscalls to limit the length of CPU burst so the
processes remains in the higher priority queue

6.23
a. the running process will have highest priority than all waiting processes, the earlier processes in the ready queue
have higher priority than later processes, effectively FCFS
b. the running process will be preempted by a new coming process, without new processes the running process cannot be

preempted by waiting processes, new processes have the highest priority, a preemptive LIFO queue

6.24
a. no discrimination at all
b. no discrimination at all
c. short processes will be assigned higher priority by their short CPU burst

6.25
// skipped, refer table 6.22

6.26
HIGH_PRIORITY_CLASS and HIGHEST

6.27
// skipped, refer table 6.23

6.28
// thanks https://en.wikipedia.org/wiki/Completely_Fair_Scheduler#Algorithm
both CPU bound:
vruntime of A will decay faster than B
A IO, B CPU:
A will have a slower increasing value of vruntime which also decays faster
whenever A is ready B will be preempted after its time quantum
A CPU, B IO:
A will have a faster increasing value of vruntime which decays faster
potentially both A and B will have vruntime float around a limit, which depend on their burst time and nice value
which ever have a lower stable vruntime value has higher priority

6.29
as stated in the text, when a higher priority process requires use of a shared resource currently locked by a lower
priority process, the lower priority process temporarily inherits the higher priority to prevent being preempted by
another process
a possible solution is to transfer the shares of the blocking high priority process to the locking lower priority
process so the lock can be released eariler

6.30
priority of a process in rate-monotonic scheduling is fixed, when there are a huge number of processes CPU utilization
of rate-monotonic scheduling is limited to ~70% but EDF may be able to fully utilize the CPU

6.31

a. P1 has higher priority
0 -> 25: P1
25 -> 50: P2 (5 remaining)
50 -> 75: P1
P2 miss deadline, cannot be scheduled by rate-monotonic scheduling

b. 0 -> 25: P1 (d = 50)
25 -> 55: P2 (d = 75)
55 -> 80: P1 (d = 100)
80 -> 110: P2 (d = 150)
110 -> 135: P1 (d = 150)

6.32

they may affect both response and turnaround time

7.1

1. a traffic jam in which cars from both sides waiting for the other to yield
2. a quarrel between two children, both waiting for the other to apologize
3. a draw by threefold repetition in chess, both players waiting for the other to make a mistake

7.2

when processes do not require and release all the needed resources at once, i.e. the resources locked by a process can
be partially released so another process can make progress

7.3

./OS/process/banker

a. 0 0 0 0
0 7 5 0
1 0 0 2
0 0 2 0
0 6 4 2

b. is safe

c. can be granted immediately

7.4

there will be no hold and wait situation in this system: any process trying to acquire any resource must first acquire
F, but F is mutual exclusive thus only one process can acquire F, a process either hold F or no resource at all, but the
resource utilization is much lower than circular-wait scheme in which resources can be acquired by multiple processes

7.5
each iteration of step 2 and 3 takes O(mn) time, each iteration marks one finish[i] as true, O(n) iterations in total
O(mn^2) operations in total

7.6
a. if the deadlocks can be completely avoided, the system will save 20 dollar per month
b. the turnaround time is increased by 20%, if the jobs are time critical that would be unacceptable

7.7
if the synchronization tools used by the process is managed by the system (mutex, semaphore, about anything except a
spin lock), the system can scan periodically over the waiting list of a mutex and detect that a process is never run
for a long time

7.8
a. no, the processes now can be preempted
b. a process may be constantly preempted by incoming new processes, never gain access to needed resources

7.9
when the state is unsafe, no process can make progress without another thread releasing its resources first, by
definition the state is a deadlock

7.10
impossible, the hold and wait condition cannot be satisfied

7.11
mutual exclusive: each intersection can only be occupied by traffics from one direction at a time
hold and wait: all the four line traffics are occupying an intersection and waiting another to be empty
no preemption: traffics cannot be removed from the intersection without passing it
circular wait: illustrated by the figure

7.12
possible, write attempts are still mutual exclusive

7.13
if only the process running do_work_one is preempted by process running do_work_two after
```
    pthread_mutex_lock(&first_mutex);
```
for e.g. fairness then the deadlock will occur

7.14
assign a unique total ordered id to the accounts, always lock the account with a smaller id first

7.15
a. circular-wait scheme incurs less overhead than banker's algorithm, no quadratic or higher complexity procedure is
    running periodically, the requests can be fulfilled or rejected in O(1) time
b. throughput of circular-wait scheme is considerably lower than banker's algorithm as some rejections in circular-wait
    scheme are totally artificial, indicating no potential unsafe state or deadlock

7.16
a. safe, ignore these new resources and the sequence is still safe
b. unsafe, after a process releasing its resources the next process in the sequence may not be able to acquire needed
    resources
c. unsafe, same to b
d. safe, the same set of resources can be allocated to the process anyway
e. safe as long as they request resources through the deadlock avoidance system
f. safe

7.17
by pigeon hole principle there will be at least one process being allocated two resources and can accomplish its job

7.18
for a deadlock each process must be holding less than necessary resources and there should be no resources available
the n processes can only be holding less than m + n - n = m resources, there should be resources available

7.19
use banker's algorithm and set max resource need to 2 for each philosopher
or simply if it's the first request by a philosopher, there should be 2 chopsticks available

7.20
assume the philosopher holding most chopsticks is holding n, there should be 3 - n available in the system

7.21
let available = [1, 1], max = [[1, 1], [1, 1]], no request would be rejected but if
    P0 hold A and request B
    P1 hold B and request A

there may be a deadlock

7.22
./OS/process/banker
a. unsafe, no safe sequence
b. safe

7.23
./OS/process/banker
a. (P0, P3, P4, P1, P2)
b. can be granted
c. cannot be granted, unsafe state and possible deadlock

7.24
processes that acquire resources will eventually releases these resources
when the programmer is careless a resource may not be released, the system may not be able to recover these resources
automatically on process exit

7.25
a state in mutex and a condvar
```
enum Bridge {
    South(u32), // south villager increment counter, north villager block on condvar
    Empty,      // both pass and change to state above or below
    North(u32), // south villager block on condvar, north villager increment counter
}
```
when a passenger reached the other end the counter is decremented by 1

7.26
the state additionally contains a traffic light which switches between two colors periodically (modified by a separate
thread or implicitly inferred from the system time by villager threads)
when the light is GREEN only south villagers can pass
when the light is RED only north villagers can pass

7.27
./OS/process/vermont-bridge

Programming Project
./OS/process/banker

8.1
1. logical address has a fixed range defined by the architecure (2^32 in x86, 2^48 in x64), while physical address has

range varying from individual system to system according to the hardware installed
2. logical addresses can be non continuous due to segmentation and paging, but physical addresses are always linear
and continuous

8.2
pros:
code section cannot be modified, the same program can be loaded into the memory once and executed by several
processes without race condition
cons:
cannot think of anything, the PCB is slightly bigger

8.3
so a logical address can be easily partitioned to a page index and a page offset
if the page size is 2^n, the lower n bits of the logical address will be the page offset

8.4
assume N bit words
a. 16 + N
b. 15 + N

8.5
the two blocks of logical addresses will point to the same block of physical addresses, from the view of the program if
the two blocks are read-only they appear to be two blocks with the exactly same content
copying a block of content can be implemented as changing the entry of the target logical address in the page table to
the source physical address, but the two blocks must be read-only from now on or write to one address will be visible
from another

8.6
a segment table
the standard C library may be loaded to the memory once and multiple processes may have different base-limit pair in
their segment table pointing to the same physical memory of standard C library

8.7
a. a segment table loaded to a specific register and searched by the CPU
b. a page table loaded to a specific register and searched by the CPU

8.8
a. all memory access are directly performed to physical addresses with key 0

b. the only user should have root privilege, logical memory addresses can be accessed with key 0
c. if a block of memory is assigned to a process, along with its entry in page table the OS should also store the key
    so that only the process can access the memory, unauthorized access will be trapped to the CPU
d. no difference to part c
e. key is stored beside page table entry
f. key is stored beside segment table entry

8.9
internal fragmentation is inherent to a partition system, e.g. memory in the system can only be allocated a fixed-size
block at at time so any allocation not a multiple of the block size will contain a portion of memory that's occupied
from the perspective of the OS but not usable by the user program
external fragmentation is caused by the variant size of allocations in a system, rapid load and unload of processes may
break the free memory space into small pieces, which has enough combined size to house a large allocation of a new
process but none of them alone is large enough

8.10
// refer chapter 7 of CSAPP
two main tasks are symbol resolution and relocation
when generating relocatable object files compiler should also generate symbol tables so the linker can resolve each
reference to exactly one symbol
then the same type of sections in the object files (read-only data, global variable, etc.) are merged into one, symbol
references are relocated to correct run-time addresses directed by the relocation entries generated by the assembler

8.11
first fit:
    185, 100, 150, 200, 17, 125
best fit:
    300, 100, 350, 17, 10
worst fit:
    300, 600, 350, 200, 750, 125
    300, 600, 350, 200, 635, 125
    300, 600, 350, 200, 135, 125
    300, 242, 350, 200, 135, 125
    300, 242, 150, 200, 135, 125

(cannot allocate 375)

8.12
a. each process is assigned a continuous block of memory on creation, processes may have to claim the amount of memory
they want before creation
heap allocations then can be direct or managed by an allocator to minimize external fragmentation
if memory locations cannot be relocated in run-time, dynamic memory allocation beyond a certain limit is impossible,
processes cannot be moved to a bigger continuous block of memory on fly
b. the OS must allow processes to create and delete segments dynamically
c. no special treatment is required

8.13
a. allocations in contiguous memory allocation and pure segmentation scheme all ultimately have to be managed by an
allocator, and the severeness of external fragmentation is determined by the strategy used by the allocator. pure
paging does not suffer from external fragmentation as memories are always allocated as a fixed-size block, and
continuous logical memory does not have to be mapped to continuous physical memory
b. contiguous memory and pure segmentation do not incur internal fragmentation, pure paging cannot allocate memory not
of size multiple to the page size hence each allocation causes on average half a page to be wasted
c. impossible in continuous memory, straight forward in pure paging and pure segmentation

8.14
if the process does not own a page, the page identifier is not contained in its page table, referring that page will
result in a page fault and be trapped to the OS
logical page in page tables of processes may be mapped to the same frame in physical memory, but the shared memory in
most case should be read only

8.15
backing storage, usually flash memory, in mobile operating systems are not particularly bigger than the main memory,
swapping will not significantly enhance multiprogramming and may damage the flash memory which only tolerate very
limited number of write compared to magnetic disks

8.16

following 8.15, the secondary disk is usually more spacious than the built-in memory, and damaging it would not affect
the memory storing the operating system

8.17
segmentation uses about 2 usize (16 bytes on x64) for each segment id, paging without hierarchical structure for quick
access must contain an entry for each page id, possibly occupies megabytes per process

8.18
first it provides another layer of safety to memory accesses: unmatched ASID is treated the same way as TLB miss, also
ASID allows a single TLB to store virtual memory mapping for several processes, so the multiple level of TLBs do not
have to be flushed after a context switch

8.19
// thanks solution manual
a. if the process cannot be relocated on fly, the amount of stack space must be pre-determined on process creation
   the amount of stack space must be big enough to house the deepest call, so most of the time a large portion of the
   stack space is unused
b. the stack segment still have to be big enough to house the deepest calls, but it does not have to be continuous to
   the .text and .data segments
c. pure paging can handle this scheme with ease

8.20
a. (3, 13)
b. (41, 111)
c. (210, 161)
d. (634, 784)
e. (1953, 129)

8.21
a. 2^(21 - 11) = 1024
b. 2^16 = 65536

8.22
unlimited

8.23
a. 8 + 12 = 20

b. 6 + 12 = 18

8.24
// errata: http://os-book.com/OS9/errata-dir/os9c-errata.pdf
a. 2^(32 - 12) = 1048576
b. depends on the physical memory installed on the system

8.25
a. 100ns
b. 75% * (100 + 2) + 25% * 200 = 126.5

8.26
// thanks
https://faculty.psau.edu.sa/filedownload/doc-6-pdf-214a793090ffcaee487d7c0e1d5d23b0-original.pdf
paging can totally eliminate external fragmentation which pure segmentation suffers, and a long and sparse page table
can be segmented to reduce the memory occupation

8.27
with segmentation the single base-limit pair can be shared among processes, in pure paging each frame of the reentrant
module have to be mapped to a page in each process sharing that module

8.28
a. 649
b. 2310
c. segfault
d. 1727
e. segfault

8.29
when the logical address space is huge and sparse, a single level page table may be enormous, occupies unacceptable
amount of memory, by segment the page entries, lots of the entries in the top level page table can be left empty, pages
not used by the process won't have their mapping to physical memory stored in the table

8.30
if location of the four page tables are stored in registers, each memory load first loads the entry of the right page
table, then the frame location stored in that entry, two operations in total

8.31

if the segmented, hierarchical page table scheme has n level, a memory load is translated to n + 1 memory operations
when the load factor p is above 1, linked list in a single level hashed page table have average length p, a successful
access has to traverse p / 2 nodes on average
when p < 2n, hashed page table is faster, otherwise hierarchical page table is faster

8.32
a. assume 4K page size
   from the 48-bit logical address <s: 13, g: 1, p: 2, offset: 32>:
      g determines the descriptor table (local or global) to be searched
      the segment pair (base limit) corresponding to the segment number s in that descriptor table is extracted
      CPU checks whether the process has the privilege to access that segment with protection field p
      the 32-bit offset is compared to the segment limit and added to the segment base, producing a 32-bit linear address
   from the 32-bit linear address <p1: 10, p2: 10, d: 12>:
      the top level page table is accessed with p1, returns the location of the second level page table
      the second level page table is accessed with p2, returns the location of the physical frame
      d is added to the base of the physical frame, producing a 32-bit physical address
b. segmentation + hierarchical page tables, 8.26 and 8.29 combined
c. accessible physical memory is limited to 32 bit or 4GB, but the main reason IA-32 is not adopted by every single
   manufacturer is competition for the sake of possible market domination from IBM etc.

8.33
// skipped, why is this even a programming assignment

9.1
a page of virtual memory is not in physical memory when the process is trying to access it
1. the page fault is trapped to the OS
2. the process is switched out, its registers and states are saved to its PCB
3. if the access is illegal (e.g. out of bound), a signal is thrown to the process and most of the time the process is
   terminated on next execution
4. assign a free page if any to the process, call the backing store device to read the target frame to the page, put
   the process in the waiting queue of that device
   if no or few free pages are available in the system some processes are suspended, all of their pages are freed
5. the CPU is allocated to other processes in ready queue before the read is complete
6. the IO device interrupt the CPU, the currently running process is paused and saved to PCB

7. the page table is corrected, the page caused the page fault is now in memory
8. once the process is allocated CPU again, it continues execution from the instruction which caused page fault

9.2
a. n, each page must be brought into physical memory on first use
   exactly n page faults occur when the first n references are to each page once, and all following references are to
   the same page
b. if m >= n, all n pages can reside in the physical memory, n page fault at most
   if m < n, with FIFO replacement algorithm each reference may cause a page fault, p at most

9.3
9EF -> 0EF
111 -> 211
700 -> not in memory
0FF -> not in memory

9.4
a. 4, no anomaly
b. 1, anomaly
c. 5, no anomaly
d. 3, anomaly

9.5
1. the ability to translate logical memory to physical memory through a page table and possibly multiple level of TLB
2. a large secondary storage to store swapped out processes and pages
3. the ability to restart the execution of an instruction after a page fault so that the memory is not left at some
   inconsistent state

9.6
seeking time on average is 60/3000/2 = 10 millisecond
transfer time of a page is 10^3 / 10^6 = 1 millisecond
99% instructions do not access another page, 1 microsecond
1% * 80% = 0.8% instructions access another page in memory, 2 microsecond
1% * 20% * 50% = 0.1% instructions access another page not in memory and the replaced page is clean
   the page has to be sought and transferred to memory, 11002 microsecond ignoring time spent on interrupt handling
1% * 20% * 50% = 0.1% instructions access another page not in memory and the replaced page is dirty
   the page has to be written to the dram first, 22002 microsecond

34.01 microseconds on average

9.7
assume the page size is measured in int (word), matrix is stored in row major
a. each 4 iterations reference two new pages, the process is in constant using and will never be replaced
   100 * 100 / 4 = 2500 page faults
b. each 400 iterations reference two new pages
   100 * 100 / 400 = 25 page faults

9.8
./OS/memory/replacement
1 physical frames
   FIFO: 20 page faults
   LRU: 20 page faults
   OPT: 20 page faults
2 physical frames
   FIFO: 18 page faults
   LRU: 18 page faults
   OPT: 15 page faults
3 physical frames
   FIFO: 16 page faults
   LRU: 15 page faults
   OPT: 11 page faults
4 physical frames
   FIFO: 14 page faults
   LRU: 10 page faults
   OPT: 8 page faults
5 physical frames
   FIFO: 10 page faults
   LRU: 8 page faults
   OPT: 7 page faults
6 physical frames
   FIFO: 10 page faults
   LRU: 7 page faults
   OPT: 7 page faults
7 physical frames
   FIFO: 7 page faults
   LRU: 7 page faults
   OPT: 7 page faults

9.9
// thanks solutions manual
the valid / invalid bit can be used as a reference bit

when the reference bit has to be cleaned e.g. in second chance algorithm, the valid bit is set to invalid instead, next
access to the page will be trapped to the OS even the page is in physical memory, OS then set the bit to valid
under this scheme a memory access would be trapped to the OS even if the page is in physical memory, dramatically
increasing the cost of memory load

9.10
it's not optimal
if a sequence of references can be scheduled with k page faults with n physical frames, the same schedule can be applied
to any system with n + p physical frames by simply ignoring the additional p frames, hence a page-replacement algorithm
that experiences Belady's anomaly cannot be optimal

9.11
FIFO:
    segments are allocated physical memory one beside another, when segments must be replaced for a new allocation, the
    old segments are swapped out one by one from an most_early_segment index until the consecutive free memory is large
    enough to house the new segment, the physical memory is managed must like the circular buffer in section 5.7.1
    compaction is performed from time to time if the segments can be rearranged
LRU:
    // thanks solutions manual
    select the oldest segment among the large enough ones (including the holes on two sides of them), if there's no
    large enough segment a chunk of oldest neighboring segments by some criteria are swapped out
    if the segments can be rearranged the segments can be compacted and sorted by last access time, the first
    replacement after compaction is exactly the same to FIFO

9.12
a. either the processes are thrashing or all of them are IO bound, without additional information e.g. page fault rate
    it's unsafe to increase the level of multiprogramming of the system, if processes are thrashing the paging system
    may be improved so that processes are given enough pages or swapped out
b. some CPU bound process are busy doing its job, CPU utilization cannot be improved any further by increasing the
    level of multiprogramming

c. processes are either idle or waiting for some non IO events, increasing the level of multiprogramming is a valid
option to improve CPU utilization

9.13
it's impossible if the base and limit are not on a page boundary, normally a page table cannot limit accesses to only
part of a page

9.14
TLB miss, no page fault:
the page is loaded but replaced by the TLB replacement algorithm later
TLB miss, page fault:
the page is not loaded into physical memory and on backing store
TLB hit, no page fault:
an ideal scenario, quick memory load
TLB hit, page fault:
incoherent cache, should be impossible, when a page is replaced its entry in TLB must be flushed, the system or the
hardware has a bug

9.15
a. to Blocked state, the page causing fault must be loaded into memory by an IO device and the process has to wait
b. no change, TLB is hardware and should be transparent even to the OS
c. no change, a success memory load typically does not go through OS

9.16
a. with pure demand paging the only way to brought a page into physical memory is through page fault, all memory
accesses to the program and the first locality will cause page fault, page fault rate would be around 100%
b. the process during execution moves from one locality to another, when it first enters a new locality it has to
bring the pages of the locality to physical memory, subsequent accesses in the locality will not cause page fault
c. 1. more pages can be assigned to this process (may be handled automatically after the OS observed a higher than
threshold page fault rate) taken from other processes
2. no action is taken, this process will fault from time to time during execution
3. there's no free memory in the entire system and this process is chosen to be suspended, it's swapped out to
backing store and its pages are reassigned to other processes

9.17
when a page is marked as copy-on-write, copying this page by e.g. fork() syscall will not duplicate it in the physical
memory but only create an entry in the page table pointing to the same frame, only when the process tries to write to
the page the associating frame on physical memory will be duplicated to a new free frame
with copy-on-write a fork() syscall does not have to immediately copy the entire memory space of the parent process for
the child process, if the child process calls exec() syscall right after fork() the memory copy would be unnecessary
the CPU must be able to understand the copy-on-write bit in the page table so while the attempt to write to a copy on
write page will be trapped to the OS, read attempts will bypass OS and be handled as usual, otherwise the memory access
latency would be unacceptable

9.18
page id: 2715, page offset: 2816
the MMU searches the TLB by the page id, if it found an entry of physical frame location the frame location is combined
with the page offset to produce a physical address, otherwise it start searching through the in memory page table whose
location is stored in a special register
if the entry is found in the page table is found the load is performed as above, otherwise a page fault is trapped to
the OS and the OS will try load the page from the backing store and update the page table, whose content is later loaded
into the TLB by the hardware

9.19
100(1 - p) + (0.3 * 8 * 10^6 + 0.7 * 20 * 10^6)p <= 200
(16400000 - 100)p <= 100
p < 1/163999 = 0.00000610

9.20
since each user thread is mapped to a kernel thread, one thread blocking on IO will not affect the other threads, but
the page table is shared among these threads and should be protected somehow or there's possible race condition

9.21
./OS/memory/replacement
3 physical frames
FIFO: 17 page faults

LRU: 18 page faults
OPT: 13 page faults

9.22
a. E12C -> 312C
3A9D -> AA9D
A9D9 -> 59D9
7001 -> F001
ACA1 -> 5CA1
b. 4000
c. anything but page 3, 7, A and E

9.23
a. the system is running out of free frames, its trying to reset the reference bit faster to allow more pages to be
replaced when necessary
b. the system has plentiful of free frames

9.24
when a page is accessed periodically but infrequently, LRU will keep it in the physical memory but LFU will evict it,
possibly make rooms for more frequently accessed pages hence increase hit ratio
when a page experienced a burst of access and is never going to be accessed, LRU will quickly evict it but LRU will keep
it in the physical memory until its access count decays

9.25
when the program initializes a large set of pages, do something else and only then works on the pages, LRU will evict
these pages from the memory so when the process works on them they have to be brought into memory by page faults, in
contrary MFU possibly will keep them in the memory
in almost all other cases LRU are better than MFU

9.26
a. a victim page is chosen from resident pages by FIFO, while the victim page is written out to the backing store a
free frame is chosen from the free frame pool by LRU, the desired content is read to the chosen free frame then
the page table is updated, when the resident page is written out it's frame is added to the free frame pool
b. the same frame is reused directly from the free frame pool, the page table is updated without any IO operation

c. a single user system, every new allocation will cause the current loaded page to be written out, the system is
    unlikely to be functional at all
d. FIFO replacement algorithm

9.27
a. no help, the processes are thrashing, a faster CPU cannot manage memory any better than the old one
b. no help, as long as the system has enough disk space to swap out the pages additional disk space has no effect on
    thrashing
c. more processes will be there demanding memory, the state of thrashing would be even worse
d. may help, by decreasing the number of processes, the system requires less memory and thrashing may be resolved
e. may help, if the main memory is increased to a point that all pages of thrashing processes can be resided, the pages
    will no longer be swapped in and out constantly
f. no help, as long as there's no enough memory to reside all these processes pages will go back and forth between
    memory and backing store, faster disk only makes the processes spin faster in thrashing state
g. no help, prepaging will not change the number of pages in a working-set of the program, thrashing still will occur
h. no help, thrashing may be worsen as there's more internal fragmentation in the system, more memory are wasted and
    less can be allocated to processes

9.28
first the page containing the first instruction of the program is brought into physical memory by a page fault, then
the program tries to access a location storing another memory location, the indirect memory access will cause another 2
page faults
the third page fault will replace the page storing the first instruction if the replacement algorithm is FIFO or LRU and
the page is not locked, the process will start to thrash

9.29
this algorithm is almost the same as second chance replacement except that reference bit of each page is reset on its
individual period instead of a global clock
second chance replacement algorithm can be implemented above this mechanism, also the system can reclaim the pages of

a low priority process more frequently, but being a approximation of LRU the hit ratio of this algorithm cannot be
better in general

9.30
a. the initial value of the counter is 0, before a page is loaded into a frame the number of pages associated to a
    frame is 0
  the counter is incremented when a frame is replaced with a new page, a page not loaded to the frame before
  the counter is decremented when a page recently being loaded into it is loaded into another frame
  the page currently in the frame with smallest counter is replaced
b. ./OS/memory/replacement
  CNT: 15 page faults
c. OPT: 11 page faults

9.31
80% * 1 + 18% * 2 + 2% * (1 + 20000) = 401.18 microsecond

9.32
a process has no enough pages to load all of the pages in its working set, causing page faults constantly as a result
the system can record and detect the frequency of page fault of a process, if page fault rate is high the process is
thrashing
more page frames can be assigned to this process or this process has to be suspended

9.33
it usually cannot, whenever a program is working on a set of data in memory it commonly does it in a particular function
or loop, so neither its active code nor data section will change during the working set of data, but there may be rare
cases in which a program works on the same set of data in two different branches of deep and distant function calls,
only move from one branch to another occasionally

9.34
if Δ is too small, the working-set window cannot contain a whole locality, processes will be assigned fewer page frames
than necessary and will start thrashing
if Δ is too big, processes will be assigned too many page frames than necessary, the system will run out of memory
sooner, level of multiprogramming will decrease

```
9.35
Request 6
    512
        256
            128
                64
                    32
                        16
                            8
                            6:8
Request 250
    512
        250:256
            128
                64
                    32
                        16
                            8
                            6:8
Request 900
    // not enough memory, rejected
Request 1500
    // not enough memory, rejected
Request 7
    512
        250:256
            128
                64
                    32
                        16
                            7:8
                            6:8
Release 250
    512
        256
            128
                64
                    32
                        16
                            7:8
                            6:8
// other two releases are not allocated at the first place
```

9.36
a working set for each thread, each thread may be accessing different data and executing different function

9.37
the single cache must be shared by all the CPUs, allocation of a kernel object must be protected by mutex and block,
that means process creation, opening files, memory mapping and nearly all syscalls can only be processed one at a time,
diminishes any benefit multiple processors may provide
either each CPU should have their own object cache, or the system has to be asymmetric (i.e. different tasks and cache
are handled on different CPU)

9.38
page size can be decreased to reduce internal fragmentation or increased to reduce the size of page table and the time
and the number of IO operations to load a data set, according to the need of a process
the OS must monitor memory accesses in more details, the translation from logical memory to physical memory using the
same entry in TLB may be illegal or not depending on the current size of the page in that entry

9.39
./OS/memory/replacement

```
1 physical frames
   FIFO: 91 page faults
   LRU: 91 page faults
   OPT: 91 page faults
2 physical frames
   FIFO: 82 page faults
   LRU: 82 page faults
   OPT: 63 page faults
3 physical frames
   FIFO: 66 page faults
   LRU: 72 page faults
   OPT: 49 page faults
4 physical frames
   FIFO: 58 page faults
   LRU: 59 page faults
   OPT: 41 page faults
5 physical frames
   FIFO: 50 page faults
   LRU: 51 page faults
   OPT: 33 page faults
```

6 physical frames
  FIFO: 45 page faults
  LRU: 46 page faults
  OPT: 27 page faults
7 physical frames
  FIFO: 37 page faults
  LRU: 35 page faults
  OPT: 21 page faults

9.40
./OS/memory/collatz-winapi

9.41
./OS/memory/mmu

10.1
even in single-user environment IO may still be issued and waited asynchronously (e.g. in Node.js), a better disk
scheduling algorithm can resolve a batch of IO requests faster so the user process has to wait for less time

10.2
an IO request to a middle cylinder will be handled next under SSTF if it's the closest cylinder inside or outside the
read-write head, but the read-write head cannot move beyond the innermost or outermost cylinder

10.3
the location of sections on a track is managed by the disk controller, not fixed or exposed to the driver software
SSTF: the distance to a sector have to be calculated giving current cylinder & angle of read-write head
SCAN: IO requests on the same cylinder are sorted by their angle
C-SCAN: same to above

10.4
if all IO requests are handled by a single disk or controller, processes waiting for these requests will block longer,
resulting in a lower global throughput

10.5
reading a file by swap space raw IO should be faster than traversing the file system, but swap space occupies extra disk
space and may need to be initialized if this process is the first to use it in the system

10.6
there's always disasters big enough to wipe all backups so no

10.7
a. 1. 512 / (512 / 5 * 2^20 + 0.015) = 33.913 Kbps
   2. 483.019 Kbps
   3. 4.651 Mbps
   4. 4.977 Mbps
b. 1. 0.00647
   2. 0.0943
   3. 0.930
   4. 0.995
c. t / (t / (5 * 2^20) + 0.015) / (5 * 2^20) >= 0.25
   t >= 26214.4 bytes
d. // thanks solutions manual
   K, where K is the block size of the disk
e. cache: 2.237 bytes
   memory: 1.678 bytes
   tape: 40 Megabytes
f. depends on how the tape device is used
   if it's treated as a back up storage of disk then it's a sequential-access device, if it's used as a secondary
   storage (despite of its long access latency) then it's a random-access device

10.8
in a burst of small and random accesses, if the requests are uniformly distributed on all the disks in RAID 1 they can
be handled in parallel, while in RAID 0 they can only be handled sequentially, since the accesses are small the faster
transfer time in RAID 0 will be dwarfed by the long seeking time

10.9
a. SSTF: if there's IO requests to the cylinder around the current position of read-write head constantly, an IO
   request to the inner or outermost cylinder may never be served
   SCAN: if there's an endless stream of IO requests to the same cylinder, the read-write head will stuck at that
   cylinder, requests to other cylinders will never be served
   C-SCAN: same to above
   LOOK: same to above
b. handle one IO request on a cylinder at a time, consequential requests to the same cylinder is handled FCFS

c. a single process generating an endless stream of IO requests can block every other processes which are waiting on IO
    in the same system
d. 1. when the throughput is the only concern e.g. in a batch system
    2. when some IO requests are generated by high priority processes
    3. in a critical situation e.g. during power outage and the system is on limited emergency power, the system should
        write out all processes to swap space and that IO request should be handled with top priority

10.10
SSD has almost uniform seeking time and no read-write head, there's no point to optimize the read-write head movement by
a complicated scheduling algorithm like SSTF, also FCFS ensures fairness

10.11
./OS/storage/disk-scheduling
FCFS: 13011
SSTF: 7586
SCAN: 7492
C-SCAN: 9917
LOOK: 7424
C-LOOK: 9137

10.12
a. d = at^2, t = (d/a)^(1/2), a is constant
b. 1 = x + y
    18 = x + 70.70y
    17 = 69.70y, y = 0.244, x = 0.756
c. ./OS/storage/disk-scheduling
    FCFS: 90.058ms
    SSTF: 65.345ms
    SCAN: 68.873ms
    C-SCAN: 78.277ms
    LOOK: 66.617ms
    C-LOOK: 70.052ms
d. 26.0%

10.13
a. half a round, or 60 * 1 / 7200 / 2 = 0.0167s
b. 16.7 = x + yL^(1/2)
    L = 4269.873

10.14
pros:
    faster random-access speed and transfer speed, no need for complicated scheduling algorithm, more tolerant to
    disasters, uniform seek time, can be put to sleep when idle (the disk must spin whenever power is on)
cons:
    lower expected lifetime, limited writes, more expensive per gigabyte

10.15
ignore transfer time or assume transfer time is proportional to the distance traveled, ignore rotation latency, both
initial position p and IO request r are uniformly distributed
let e be the innermost cylinder

```
    SCAN:   int(p, 0, e, int(r, 0, e, d(p, r)))
        where d(p, r) = r - p if r >= p
                      = (e - p) + (e - r) = 2e - p - r if r < p
        = int(p, 0, e, (2e - p + 2e - 2p)p/2 + (e - p)^2 / 2)
        = int(p, 0, e, 2ep - 3p^2/2 + e^2/2 - ep + p^2/2)
        = int(p, 0, e, ep - p^2 + e^2/2)
        = 2e^3 / 3
    C-SCAN: int(p, 0, e, int(r, 0, e, d(p, r)))
        where d(p, r) = r - p if r >= p
                      = (e - p) + e + r = 2e - p + r if r < p
        = int(p, 0, e, (2e - p + 2e)p/2 + (e - p)^2/2)
        = int(p, 0, e, 2ep - p^2/2 + e^2/2 - ep + p^2/2)
        = int(p, 0, e, ep + e^2/2)
        = e^3
```

C-SCAN have longer expected response time
when requests are sparse C-SCAN may suffer from greater response time variation, about half of the requests must wait
for the read-write head to go back to cylinder 0 and start over again, but when the IO queue is crowded and higher
rotational latency C-SCAN is more fair and has lower variation as requests to the other end have to wait all the other
requests elsewhere to be served

10.16
a. SSTF would be far better than all the other algorithms, other algorithms must go towards a fixed direction so cannot
    focus on a small area
b. SSTF but waits for a small period of time before leaving the hot spot, so the upcoming requests to the small area
    have a higher chance to be handled rapidly

10.17
a. one block, the block of data
b. 9-10 blocks, 7 data blocks and 2-3 parity blocks

10.18
a. RAID 5: read operations on a single block access only one block on a single disk
RAID 1: same to above
b. RAID 5: contiguous blocks are interleaved to different disks and can be read in parallel
RAID 1: contiguous blocks are stored in the same disk, can only be read sequentially

10.19
RAID 1: each write operation writes to a single disk
RAID 5: each write operation writes to at least 2 blocks on 2 different disks, global throughput will be lower than
RAID 1, but write to contiguous blocks can be handled in parallel if there are enough disks

10.20
according to 10.18 and 10.19, file that is frequently read should be stored in RAID 5, while file that is modified by
processes should be stored in RAID 1

10.21
a. once per month, 750000 / 1000 = 750 hours = 31.25 days
b. // thanks solutions manual
MTBF = 24 * 365 * 1000 = 8760000 hours
nothing is given about the expected lifetime
c. warranty of a product is decided by financial consideration and legal requirement

10.22
sector sparing:
less available space, a hard failure of a sector would not reduce the number of available disk space so the disk can
maintain the labelled spec for a longer period of time
sector slipping:
less available space, a failed sector will always be mapped to the next available sector, the failure handling
controller is more complicated, an entire track have to be read and write to different position on hard failure

10.23
when the secondary storage is backed by RAID 5, the OS may want to ensure that a frequently accessed file has its blocks
scattered around all the disks so contiguous read can be performed in parallel

the OS may also require a large file to be stored in the same or close cylinders to minimize the seeking time

10.24
./OS/storage/disk-scheduling
FCFS: 8222052
SSTF: 8110
SCAN: 6884
C-SCAN: 9997
LOOK: 6884
C-LOOK: 9995

11.1
auto reset system may be useful on computers shared by anonymous public users, e.g. in a campus library or internet
cafe, they can use a set of pre-installed softwares while have no access to system files or other users' files
systems that do not reset on user logout is used when the user (partially) owns the computer

11.2
if the system does not provide the API to read / write different type of files according to their content, all user
programs have to parse the files by their own codes or (possibly different) third-party libraries, an approach more
bug-prone
a typed file API on the other hand may be misused or show surprising behavior from time to time, e.g. the difference
between wb and w on some old platform, programmers have to be more careful than dealing a uniform binary file API

11.3
same to above, in practice most files are either text or records of fixed size, glibc provides both blocked IO and
formatted IO, the same set of functionality can be built on a stream byte API, the system programmer has to decide
whether the system should provide these API out of box at the expense of bigger image size

11.4
the real file name can be modified to include the path to that file, for example a file log.txt under directory /etc/
can be named /etc/log.txt, as long as each file has a unique path they will have a unique name in the single-level
directory structure, then an external file explorer can simulate the multilevel directory structure from their file name

if the file name is limited to 7 characters, paths would be to long to be included in file names

11.5
without open() syscall, each read and write syscall must take file path instead of file descriptor as argument, each
call to read() and write() must first traverse the file system to find the entry of the file, either the entire file
system must be loaded into memory on startup or read() and write() syscall will be much slower
open() caches file entry in a table, without close() the OS won't know when to remove these entries from the table and
the table will eventually fill up

11.6
// thanks solutions manual
a. if the structure of the subdirectory is stored in the directory file instead of the file system, by modifying the
    subdirectory the users owning the subdirectory may include files which they do not have access to the subdirectory
    and gain access to these files
b. instead of raw read and write provide another set of syscalls to manipulate directories

11.7
a. create a new group, include 4990 users in that group, set group access permission to rwx
b. create a new group, include the 5000 - 4990 = 10 users, deny access to that file by users in that group
    possible on windows platform

11.8
there will be exactly the same number of access permission entires, but the access permissions of a single user is now
specified at the same place, batch processing a user's access permissions no longer requires traversing the file system,
a user can be removed from the system much easier

11.9
even though the file is deleted, uninformed users will refer to the new file by the links as if the file is still there,
an adversary can exploit this scheme by supply a a malicious replacement and void the security of the system
link should contain a hash of the file which is updated on each access, if the file is modified or replaced since last
access through that link OS can detect a hash mismatch and inform the user

11.10

the text described a two-level scheme:
1. a system-wide open file table that stores the cached entry of the opened file, a counter of how many processes are
   accessing it (incremented on open() and decremented on close(), removed when reduced to 0), its position on the
   disk, etc., information that's independent to individual processes,
2. a process-specific open file table that stores the pointer to the system-wide open file table, the current read and
   write position into the file, the operation mode, etc.

but if processes are executed by different CPUs, the system-wide open file table must be protected by a mutex lock,
this scheme may have lower throughput than maintaining a full-fledged open file table per user

11.11
pros:
   once the lock is acquired the protection is absolute, no other user can gain access to the same file
cons:
   the file system is now more dangerous to use, read() or write() syscalls may block a process indefinitely, a
   misbehaving process has the potential to block a big portion of the system by deadlocking while holding the lock of
   a widely shared file

11.12
sequential:
   video / music player, video game
random:
   database, word processor, text editor, web browser

11.13
pros:
   users no longer have to manually call open(), read() and write() can skip the directory structure by hashing the
   file name into a hash table
cons:
   file in the open file table lives much longer than necessary, fewer processes can run concurrently

11.14
buffer the reads ahead, whenever the process calls read() the system reads much more from the file and put the bytes in
a buffer, subsequent reads and writes is served by the buffer instead of the file on the disk, the buffer is written

back to the file only when a read reads beyond the current buffer size or the file is closed

11.15
a video player that allows users to seek to a particular frame of a video, if bit rate of the video is variable the
position of that frame cannot be calculated and must be searched among the file or the index created by the system

11.16
pros:
  most systems allow such link, forbid it would be counter-intuitive
cons:
  when mount point is shared by different devices, for instance different usb storage may be mapped to the same drive
  letter, the link may be invalid or even accidentally point to a different file of the same path

11.17
stated in depth in the text in section 11.3.6
a middle ground is duplicated file with synchronization, e.g. git or other version control software

11.18
pros:
  errors that only exist on remote file system can be specified and prompted to the user, error messages will be more
  informative, the system can handle remote file system only errors in its own context
cons:
  a different set of API must be provided, or the same API for local file system is reused and behaves totally
  different when the file is located in a remote file system, leading to surprises and painful debug sessions

11.19
each write operation must be immediately reflected to the original file in the remote file system, read and write
operations cannot be buffered by the system (the user still can buffer them at their own risk), read and write
operations are much slower and suffer from network conditions more frequently

12.1
contiguous
a. 202, 100 blocks are read and written to the space occupied by the next block, the new block is then inserted to the
  first free block, then the control block is updated to reflect the new length of the file

b. 2 * (100 - i) + 2, if the block is inserted to ith (0 based) place, 2 * (100 - i) blocks have to be moved
c. 2, 1 block inserted to the end, the control block updated
d. 1, the control block is updated to point to the second block with length - 1
e. min(), either all blocks before it or all blocks after it is moved
f. 1, the control block is updated to reflect the new length
linked
a. 2, the new block contains a pointer to the former first block, the control block is updated to point to the inserted
    block
b. i + 2 for i > 0, the block is inserted as the new ith block (0 based), i blocks are traversed, the i-1th block is
    updated and written back, the new block is inserted
c. 102, 100 blocks are traversed, the last block is updated and written back, the new block is inserted
d. 2, the first block is read, the pointer to the second block is extracted from first block and updated to the control
    block
e. i + 2 for 0 < i < 100 (0 based), the i + 1 blocks are traversed, the i-1th block is updated to point to the i+1th
    block
f. 100, 99 blocks are traversed, the 98th block now points to null
indexed
a. 2, the index block is updated, the new block is written out
b. same to above
c. same to above
d. 1, the index block is updated
e. same to above
f. same to above

12.2
even if the system can track and manage these mounts, a user program either cannot or won't be bothered to track all
these duplicated mounts, operations which are supposed to be performed on each file once may be performed multiple times
on files in that file system
if the system in addition allows cascading mount, the same file system can be mounted to a subdirectory of itself,
possibly creating a loop in the file system

12.3
otherwise a special tag must be contained in each block to indicate whether it is free, and the entire disk must be

scanned on startup to initialize the bitmap, when the disk is spacious this initialization will take several hours

12.4
1. whether the file will be accessed sequentially or randomly, linked file do not support random access well
2. the maximum size of the file, huge file usually cannot be allocated contiguously
3. if the file will be accessed in its entirety frequently, storing it in contiguous space may increase effective
   transfer rate

12.5
same to the cluster scheme described in the text, more internal fragmentation but free from external fragmentation

12.6
read can be buffered ahead so subsequent reads can be served by the buffer instead of causing another IO operation,
write can be buffered so multiple write to a single block can be applied once instead of causing an individual IO
operation each time
cache is more expansive than disk and a system with more cache will suffer more significant inconsistency between memory
and disk after disasters

12.7
pros: the system occupies less memory when the functionality are not used
cons: user will experience noticeable delay first time calling these syscalls

12.8
each file system implements an interface defined by the VFS specification, OS can handle syscalls from user programs by
calling VFS functions from whatever file system mounted at the path transparently

12.9
a. exactly the same to linked file, no external fragmentation, bad at random access
b. the OS must choose between external fragmentation and long links of scattered file extends by ignoring free holes
   under a certain size or not, and it would difficult for the system to decide how much space should be allocated to
   the next extend of a file and may delegate the decision to the user
c. no external fragmentation and the efficiency of random access can be optimized according to user requirements

12.10
contiguous: highly efficient sequential and random access, sequential access can be performed in batch
linked: random access has Ω(n) complexity
indexed: efficient sequential and random access

12.11
in ordinary linked allocation accessing ith block of a file requires Ω(n) IO operations, with FAT the links can be
traversed in one or fewer blocks, provides better support to random access

12.12
a. if the file system can obtain a list of all the blocks assigned to it from the OS, it can reconstruct the free-block
   list by deducting the blocks allocated to files from the entire list of blocks
b. thanks http://matthews.sites.truman.edu/files/2017/01/chapter12.pdf
   in worst case each directory on the path contains enormous number of files, forcing any access to go through the
   triple indirect pointer, assume the block of the file entry in the directory can be located immediately by a hash
   table
   1 - 5: read inode or root, read the 3 indirect index blocks, read the file entry block
   6 - 10: same to above for /a
   11 - 15: same to above for /a/b
   16 - 17: read inode of /a/b/c and the direct block containing its content
c. doubly link all the free blocks as a circular list, storing pointers to previous and next blocks in the header, each
   single free block can be used as the new first free block pointer after a memory failure

12.13
small file can be allocated small blocks to minimize internal fragmentation, bigger file can be allocated blocks of full
size to minimize the necessary disk operations for sequential access
alongside the pointers to the next block, the size of the current block must be stored in each free block, or blocks of
different size must be managed in different lists, blocks of smaller size may need to be combined to form a block of
bigger size on demand

12.14
write cache: after a crash data in the write cache yet to be written out is lost, causes inconsistency between control
   block and data

free block bitmap, block index and hash table: by using complicated structure the critical section of an IO operation
 is longer, the system is more vulnerable to crash and data inconsistency
grouping and counting: more difficult to reconstruct the free list after a crash

12.15
a. how is memory address translation part of the file system
b. contiguous: 1, the location of block 4 can be computed from the location of block 10
 linked: 6, the links must be traversed one by one from block 10 to block 4
 indexed: 2, read the index block and then block 4

12.16
8KB / 4B = 2048
(12 + 2048 + 2048^2 + 2048^3) * 8KB = 64.031TB

12.17
a reverse map from block number to files can be built by scanning the entire file system, then blocks can be swapped
to move all free blocks to the end of the disk space
1. doing so requires hours of time, during which normal operation cannot be served or is significantly delayed
2. on certain disk devices recompacting and relocation consumes limited read and write tolerance
3. when programmed carelessly the entire recompacting session is a critical section, crash in between will leave the
 disk in an inconsistent state

12.18
1. each time a file system is mounted or unmounted, the cache entries with a corresponding prefix must be invalidated
 to hide or expose the underlying local directory
2. on certain error condition returned by the remote file system calls, the cache entries must be invalidated, e.g.
 a file is moved or deleted on the server
3. the cache may even be flushed periodically

12.19
an OS recovering from a crash can inspect the log to see what IO operation is yet to be performed or halfway done, then
revert to a safe state and apply these operations anew until the log is emptied and the user programs agree with the
OS about the disk state

12.20

a restore operation only read and copy files from two mediums instead of all mediums from the last full backup, but each
backup occupies more space than the scheme in section 12.7.4

12.21

```
$ echo foo > file1.txt
$ ls -li file1.txt
32932572275255899 -rw-rw-rw- 1 *** *** 4 Dec  9 19:41 file1.txt
$ ln file1.txt file2.txt
$ ls -li file2.txt
32932572275255899 -rw-rw-rw- 2 *** *** 4 Dec  9 19:41 file2.txt
$ cat file1.txt
foo
$ cat file2.txt
foo
$ echo baz > file2.txt
$ cat file1.txt
baz
$ rm file1.txt
$ ls
bin  file2.txt  file3.txt
$ strace rm file2.txt
// ...
unlinkat(AT_FDCWD, "file2.txt", 0)      = 0
// ...
$ ln -s file3.txt file4.txt
$ ls
bin  file3.txt  file4.txt
$ ls -li file*.txt
39687971716484175 -rw-rw-rw- 1 *** *** 4 Dec  9 19:41 file3.txt
83035118130003454 lrwxrwxrwx 1 *** *** 9 Dec  9 19:58 file4.txt -> file3.txt
$ echo foo > file4.txt
$ cat file3.txt
foo
$ rm file3.txt
$ cat file4.txt
cat: file4.txt: No such file or directory
```

the command is translated to

```
   cat file3.txt
```

and the file referred by the link no longer exist

13.1
pros:

hardware implementation executes faster than software implementation in kernel or user space
the functionality is better encapsulated, user program or the OS is not even aware of its existence
the code or circuit of the functionality cannot be modified in any means by the host, immune to virus and malware
cons:
bugs in the implementation can only be patched by switching the hardware or flashing the EEPROM
the functionality is completely hidden from the OS, no way to cooperate
developing the functionality on hardware cost more time and expense, more difficult to debug and iterate

13.2
with a single busy bit:
the host repeatedly scan it until it's cleared, then set the data registers and set the buzy bit
the controller repeatedly scan it until it's set, serve the operation request and clear the buzy bit

13.3
interrupt-driven serial port:
the device behind the serial port rarely generates events, e.g. an sensor that captures environmental data every
few seconds, a thread should not busily wait for the event but let the OS wake it upon an interrupt
polling terminal concentrator:
when the OS is on a host dedicated to the management of the terminal concentrator(s) or when the concentrator is so
busy that generate a series of events the interrupt handler cannot deal with

13.4
the driver can poll for a fixed amount of times then ask the controller to interrupt the OS and go to sleep if no event
occurred in the period
pure polling:
a coprocessor with high frequency and small data input, a request can be served in nanoseconds, interrupt handler
cannot operate at that speed or feed the coprocessor with data consistently
pure interrupts:
a sensor reading environmental data every few seconds, process working on such device will be idle most of the time
and has no time constraint
hybrid:

a device which generates events in burst e.g. a microphone, with the hybrid strategy the events can be handled fast
and the process can be put to sleep when idle

13.5
instead of polling in a busy loop and monopolize the core, multiple IO requests can be issued concurrently, controllers
of different IO devices then can handle these requests and copy data to the supplied addresses at the same time
instead of a few registers grouped into a serial port, these controllers must be able to understand (virtual) memory
addresses and be able to modify it

13.6
faster the CPU, more cycles are wasted on polling if the system-bus and device speed didn't catch up, bigger portion of
CPU time is not doing actual work

13.7
STREAMS module is a interface with a write queue and a read queue, can be read or write in unit of bytes or messages
STREAMS driver is a device driver implemented as a series or STREAMS modules

13.8
among the devices of the same type, the ones with smaller buffer should has higher priority or they are more likely to
lose data due to buffer overflow
an interrupt indicating an error should have higher priority than normal interrupts, an error condition is usually more
urgent than normal IO operations
user should be able to assign priorities to interrupts in a certain range so they can control the behavior of the
interrupt handler while not intervene the operation of the kernel

13.9
pros:
the device registers can be accessed as normal memory addresses, the user programs don't have to follow special
protocols and explicit syscalls
cons:
the mapped addresses may be accessed accidentally by e.g. out-of-bound array accesses, a software error can
escalate to a hardware defeat through the mapped addresses

13.10
a. polling with a small buffer, the thread handling mouse inputs can be executed every few milliseconds and apply all
    mouse inputs in the buffer at once, the user won't notice the difference
b. interrupt with a buffer in kernel space, the user process should yield CPU to others during the long seek time, the
    tape drive on command will copy its content to the buffer through DMA or registers, then the kernel will wake up the
    user process and copy the content to user memory
c. same to b
d. polling with memory mapped IO, GPU works at high frequency, latency from interrupt handling may not be acceptable

13.11
user programs invoke syscalls with virtual addresses, and in most system even though memory address translation does not
go through OS for performance concern, the page table is still managed by the OS, hence the OS can translate virtual
addresses to physical addresses and request IO operations from the device with physical addresses, a whole category of
illegal access error is eliminated by the translation as memory that doesn't belong to the user process won't be present
in its page table

13.12
1. context switches, both switching out when the process it put on wait list and switching in when the process is
    waken up and allocated CPU time again
2. latency introduced by the interrupt handler, e.g. select the correct handler from a chain of handlers, dispatch the
    interrupt through indirect function call
3. generating an interrupt is more time-consuming than set up the data-out registers
4. depends on the priority of the process may have to wait for arbitrarily long on the ready list
5. the current running process may be in a critical section, the interrupt may be masked and delayed until the current
    process terminates

13.13
blocking IO:
    1. when the process can not proceed unless the IO operation is fully complete, e.g. modifying a file
    2. when the IO operation takes a long time and cannot return halfway
    3. when deadline is near, blocking IO has simpler interface
non-blocking IO:

1. when multiple jobs are to be balanced among the process itself, by non-blocking IO the process can do other jobs
   when the device is not ready
2. when the IO device is only available occasionally and the process does not want to wait for the next event
3. in async environment, blocking IO does not cooperate well with the user space task scheduler, the scheduler has
   no way to know whether the task is blocking on IO or is doing meaningful work

13.14
same to async IO (or is it an alternative description of async IO?), the syscalls return immediately and informs the
process later by an interrupt, meanwhile the process can handle other jobs (explicitly or through a user space task
scheduler), improve job balance among the single thread

13.15
the controller must understand memory translation in the same way as the CPU of the host, i.e. the content and format of
the page table, the controller may also need to manage its own TLB for performance, since virtual memory is platform
specific the controller or the driver must contain platform specific codes for a variety of platforms
the controller instead of the OS can do the memory address translation, more works are offloaded to the controller which
is the purpose of DMA from the beginning

13.16
// only message passing since shared data structure is the default control group
pros:
1. IO managers are decoupled from the kernel, the kernel data structures can be kept as internal implementation
   details and be changed without breaking the IO subsystem
2. no data structures are shared among host and devices, the system as a whole is less vulnerable to state
   inconsistency caused by a crash
3. logging is made much easier, rather than artificial log messages the message itself can be recorded or even
   rewound from the log

cons:
1. higher overhead, the message has to be copied from kernel buffer to controller memory or vice versa
2. the size of kernel and IO managers combined is larger, the same set of functionality has to be implemented on
   both sides

3. when an IO manager requires more information from the kernel after an update, extra messages must be designed
   and passed as it has no access to the data structures of the kernel

13.17
// thanks solutions manual, but the reference code is over simplified
// the hardware clock must support canceling, otherwise if the three channels are all occupied, when a new request
// arrived with notification time earlier than the earliest among the currently running 3 channels, the new request can
// not be handled and must be dropped, which is both arbitrary and unrealistic

13.18
see the difference between TCP and UDP

14.1
capability lists consist of <object, right> pairs associated with each domain, access lists consist of <domain, right>
pairs associated with each object

14.2
on deletion of a normal file, the system would simply remove its entry and add all its blocks to the free list, the
content of the file will only be overwritten when the block is reused, in between any adversary with physical access
to the system (i.e. can scan the disk bypassing the file system) would be able to recover the file
but such a scheme is not perfect, due to the nature of magnetic disks the file may still be recovered with high accuracy
after several overwrites

14.3
subset

14.4
A(x, y) ⊆ A(z, y)

14.5
a procedure can go down the stack and modify the stack content inserted by other procedures, e.g. edit the return
address of the calling procedure and invoke another malicious procedure

14.6
ring-protection

14.7
capability list or access list, the right structure now contains a counter, each access to that right decrements the
counter, when the counter is 0 the right is removed from the list

14.8
whatever data structure storing the object stores a counter alongside it, the counter is incremented when a process
gains access right to that object, decremented when the process is terminated, once the counter is reduced to 0 the
object is removed

14.9
IO operations in most system are privileged instructions, if user processes can do their own IO they may be able to
modify kernel data, e.g. the swapped out PCBs on disk, and the kernel can no longer assure its own integrity

14.10
allocate the object in a privileged segment and deny access from the user process to the segment

14.11
see section 14.3.3

14.12
no, by switching the access privileges are granted and revoked, only available to a process in domain A temporarily, by
including the process would have these privileges longer than necessary i.e. violates the least-privilege principle
also the switching right is not transitive, domain A cannot grant the switching right to other domains, by including
the rights may be transferred to any domain other than A and B

14.13
each access right entry contains a time interval, the OS only grants the right to the calling process if the system
time is in the time interval (as a result modifying system time must be privileged), the game is hidden from users and
can only be accessed from a wrapper program that checks user's privilege periodically and terminates the game process
if the user no longer have the right

14.14

// thanks solutions manual
instead of raw bytes the hardware tags each object with its type (a bit pattern) and treat them differently, the same
scheme can protect memory from being manipulated by wrong procedures as when the integer typed data is passed as a float
parameter to a procedure the hardware can detect the type mismatch

14.15
when the state of an object changes the access rights to it can be granted or revoked in one place, each object can have
a default set of access rights as a special list entry, capability of a domain is scattered all over the system thus
hard to manage

14.16
the reverse of 14.15: capability of a domain can be managed (copied, shared, transferred to other domain) with ease,
access rights to an object is scattered over multiple lists, all these lists must be traversed when the state of an
object changed

14.17
access rights of one domain no longer has to be subset or superset of access rights of another domain, rights can be
dynamically granted to or revoked from arbitrary set of domains

14.18
// referring Capability-Based Computer Systems, Henry M. Levy, 1984
// https://homes.cs.washington.edu/~levy/capabook/Chapter6.pdf
to implement the least privilege principle, the process cannot be granted all the rights it needs during its entire
lifetime, instead it's granted the capability to call certain (kernel) procedures which in turn access the system
resources in request. among those procedures some of them can be certified as trustworthy so that they gain independent
rights on their formal parameters, passed by the caller. such procedure is called a "type manager" of the type of its
formal parameter. these rights are granted universally on all objects of that specific type, in contrary to the normal
instance-to-instance capability scheme in Hydra.
Multics implements a similar facility to support cross-domain procedure calls, but capability management in Multics is
not as fine-grained as in Hydra, a process can call all procedures with higher or equal ring number

14.19
a procedure / program can only access resources explicitly given to it, an object should not unnecessarily expose its
internal structure, similar to the encapsulation principle in most OOP languages
as well as least privilege principle it can minimize the possible harm a misbehaving or malicious process can do to the
system

14.20
a. a process can access all resources of higher ring number regardless of whether it need them or not, ring-protection
   scheme by itself cannot enforce need-to-know principle
b. enforced by default, capabilities is granted instance-by-instance, on procedure calls parameters are accompanied by
   a bit vector of capabilities, without the capabilities a procedure has no access to the parameter
c. any procedure above a doPrivileged() annotation has permission to resources protected by checkPermissions(), but a
   Java program cannot directly access the memory hence a reference to an object cannot be forged, need-to-know is
   enforced not by stack inspection but the JAVA language

14.21
a JAVA thread in a less-privileged domain can gain access to privileged resources by inserting the corresponding
doPrivileged() annotation into the stack before accessing the resource, bypassing the protection provided by stack
inspection

14.22
// thanks solutions manual
role in the role-based access control system closely resembles domain in the access matrix, but different to access
matrix roles themselves are not managed by the role-based access control facility but passwords, e.g. RBAC cannot grant
a role the ability to switch to another role bypassing the password system

14.23
a user program in a system adhere to least privilege principle may only cause minimized damage to the environment
on programming error or intentionally

14.24

a user program can still cause damage within its privileges

15.1
in C reference to an array is a pointer with no length information (called "array decaying"), in a safer language like
Rust reference to an array is a slice which has length information built-in, out-of-bound access to the array can be
detected in running time and cause an exception instead of undefined behavior, in Rust references can also be declared
as immutable so its content can only be read but not modified

15.2
impossible until the intruder tries to log in with that password, after then the system can inform the user through e.g.
emails each time the account is logged in so the user will notice immediately when the intruder tried

15.3
to protect the system from rainbow table attack: even if the password is in the rainbow table the hash will be different
so an adversary cannot recover the password without computing each entry of the table with the salt appended
denote the password by p, salt by s, let H be a secure hash function, H(p || s) is computed and stored along with the
salt, on user login the system takes the password input pi, compute H(pi || s) and compare it with the stored hash value

15.4
store hashed password instead of plaintext

15.5
// referring Watchdogs - Extending the UNIX File System, Bershad and Pinkerton, 1988
// https://www.usenix.org/legacy/publications/compsystems/1988/spr_bershad.pdf
pros:

1. the watchdog is a separate user-level program, compared to a kernel module it won't crash the system on
   exception, requires less privilege, can be deployed and configured more conveniently
2. instead of an universal security scheme in the kernel, users can define their own watchdogs hence deploy their
   safety measure file-by-file

cons:

1. a watchdog, once linked to a file by a syscall, re-implements other file system syscalls like open() and read(),

such re-implementation is totally transparent to the processes accessing the linked file, once the system is
breached, the intruder can e.g. redirect write() syscalls to remote host without being noticed by the user
2. correctness of a watchdog is up to the user who implemented it while most users in practice cannot configure
a security system accurately, this scheme is more or less asking for trouble

15.6
1. according to wikipedia (https://en.wikipedia.org/wiki/COPS_(software)), COPS is mainly a signature-based detection
engine, means it cannot prevent new 0-day vulnerability and should not be trusted blindly, other security measures
must be deployed in parallel
2. COPS is a user-level program, any intruder with sufficient privilege can modify its configuration thus circumvent
its detection completely, COPS should be launched from read-only media

15.7
enable as few as possible components of the system on installation, a vulnerability in a rarely used component will not
affect every system in the network
users must be provided more privileges to enable these components, which may lead to further vulnerability and a bigger
attack surface to social networking

15.8
his behavior is clearly violating 18 U.S. Code § 1030, the interesting thing is he hadn't be shut out from the industry
for his conduct and later became a tenured professor in MIT

15.9
1. unauthorized access to accounts should be prevented, physical, human and OS
2. personal information associated with accounts should not be leaked to even employees, physical and OS
3. the system should not be vulnerable to temporary power surge and outage, physical and OS (backups)
4. backup should be stored in safe place, physical and human
5. unnoticed tempering from an employee should not be possible, mainly OS and protocol design
6. the system should be recoverable from mistake of inputs, physical and OS

15.10

1. the data is secured in a broader sense: not only within the system in which the data is protected by access control,
   it also cannot be recovered from raw physical access to the disk
2. encryption when implemented right is more bug tolerate: a wrong password cannot decrypt the data, while a system may
   grant access right to unauthorized user due to a bug

15.11
1. instant message programs, man-in-the-middle can block, insert or modify the messages: end to end encryption,
   authorization and sequential number
2. remote login, if passwords are transferred in plaintext will grant access rights to man-in-the-middle: encryption

15.12
unless the two communicating entities are located nearby (i.e. can be configured by the same administrator) or can
arrange physical or any other secure but slow contact, asymmetric key exchange is always preferred, though it's
impossible when both sides do not have a well established and verifiable identity

15.13
ke is publicly available hence cannot prove the identity of the sender
the encrypted message can only be recovered by whoever possess the private key, secure channel can be established
without a shared secret

15.14
a. if the public key cannot be derived from the secret key, keep the public key secret, publish the secret key, encrypt
   the message with the public key, others can verify the author by decrypting the message
b. normal usage
c. the above two combined with two pairs of public & secret keys

15.15
true alarms: 10 * 20 = 200
false alarms: 10000000 * 0.0005 = 5000
200 / (5000 + 200) = 3.84% alarms corresponds to real intrusions

16.1
1. type 0 hypervisor, bundled with the hardware in the form of firmware, guest OSs are usually assigned dedicated
   hardware resources (a hardware partition), supports only a few number of guest OSs or type 1 hypervisors

2. type 1 hypervisor, loaded by a boot loader in place of a normal OS, runs in kernel mode, shares hardware resources
    among guests as if the guests have dedicated ownership, manage hardware resources just like ordinary OS, achieve
    efficiency with support from the hardware
3. type 2 hypervisor, launched as a user process, runs in user mode (occasionally injects kernel mode code into the OS
    by fake device drivers), run guest OSs as normal user processes, cannot benefit from hardware support

16.2
1. paravirtualization, normal hypervisors gives guests the illusion that they are running on their own system, but
    paravirtualization provides a non-standard hardware interface and requires cooperation from the guest OSs in hope
    to share the hardware resources more effectively
2. programming environment virtualization, instead of virtualizing hardware and run guest OSs it provides a common
    interface to user programs so they can invoke the same set of functionalities on different OSs and hardwares
3. emulation, instead of providing a virtualized set of the same hardware resources as the underlying host, an emulator
    simulates a different system / hardware to softwares built for that system / hardware
4. application containment, provides virtualization of operating system instead of hardware resources, manages user
    land programs instead of guest OSs, give programs the impression that they are the only program running in the
    system

16.3
1. protects the host from the guest operating systems by another layer of indirection between them
2. guest operating systems can be suspended, cloned and restored from backups with ease
3. cloud computing server can be sold in a standardized unit regardless of the capability of the hardware
4. enables live migration

16.4
on some CPU there's no clean separation between privileged and non-privileged instructions, e.g. an instruction may have
different execution result in kernel mode and user mode, as guest OSs runs in user mode, trap-and-emulate is only
possible if all privileged instructions results in a trap to the hypervisor through the host

the VMM can perform a binary translation: when the guest executes a privileged instruction in virtual kernel mode,
VMM examines each instruction beforehand and translates them to equivalent instructions on the real machine

16.5
1. new execution mode, appears to be full-privileged to a guest OS but actually passes control to the VMM on privileged
    instructions
2. hardware support for virtual page tables, the MMU can go through both page table of the guest OS and the page table
    of the VMM, mapping virtual addresses in guest OS to physical addresses without interfere from the VMM

16.6
on virtualized system a guest OS is managed by the VMM, which can easily suspend the guest OS and copy its states to
another host, but on native system the OS must suspend itself and then copy itself to another machine, its state is not
managed by a separate entity thus may be hard to collect, e.g. trying to save the kernel CPU registers may modify the
content of them

17.1
by forwarding the broadcast packets to all ports except the source, the number of packets in all but tree shaped
networks increases exponentially, such a broadcast storm will quickly congest all the links
in a small tree shaped network a broadcast packets can reach all the hosts if forwarded by the gateways

17.2
pros:
    cache hit saves an RTT to the DNS
    functional even when the DNS is temporarily offline
cons:
    causes inconsistency when the entry on DNS is updated

17.3
pros:
    once the circuit is established, transition is much more stable than packet switching and the link is free from
    congest if transmission rate never exceeds the capacity, packets arrives in order with constant delay
cons:

extra initial cost to establish the circuit, allocated link capacity cannot be recycled when the circuit is idle,
each link on the circuit is a single point of failure
viable usage: VoIP and local live video transmission

17.4
1. a transparent distributed system should provide an interface which does not distinguish between local and remote
resources, act like a conventional centralized system
2. a transparent distributed system should provide user mobility by bringing over users' environment to wherever
they log in, allow them to access from more than one terminal machine

17.5
a. the same method to move a process from one processor to another but over a network and the opened file should be
available in both hosts or be moved along the PCB
b. if the user process runs in a virtualized environment (e.g. JVM) the process can be migrated as usual as long as
the two systems have compatible file system, otherwise the process migration cannot be transparent and the user must
explicitly program how to transfer the state and resume processing on another host

17.6
a. 1. link failure, the network link between client and server failed
2. site failure, the server is offline and cannot handle requests from the client
3. packet loss, a message is not transmitted from one site to the other
b. both 2 and 3 are possible in centralized system, a server process may be terminated by a fatal error, loop back
packets may be lost due to temporary failure in network adaptor

17.7
some active sensors broadcast their measures periodically no matter there is a host accepting their broadcasting or not

17.8
a. only if there's multiple other sites monitoring the condition of B, A can contact them
b. only if there's another link between A and B and they can confirm by switching to the new link
c. indistinguishable from B goes down
recovery from failure may cause further problems, a better solution is to switch to another server on any of the failure
listed above if available

17.9

computation migration is usually visible to user programs in term of RPC calls to execute a computation on another site
and receive the result later, while process migration can be transparent to users such that the user processes can be
move to a different site and been executed there

17.10
OSI model is quite an over specification: the session and presentation layer are not always necessary for a protocol,
in practice handshaking and data representation are up to the application, while the link and physical layer are
implemented in hardware so is not of the operating system's concern
by eliminating the presentation layer applications have to define and parse their messages explicitly and may cause
error e.g. encode a number as little endian and parse it as big endian

17.11
without congestion control doubling the speed of the system doubles the number of packets pushed to the link and router,
possibly overflows their buffer and cause packet loss, the lost packets must be retransmitted and make the congestion
more severe, a less fraction of packets transferred would be original
proper congestion control is necessary when the link may be at capacity

17.12
the same set of functionalities can be implemented with just enough hardware or outright be implemented in hardware,
greatly reduces the cost of the device and boost service stability at the same time as there's no other processes
running on the device
the downside is the device may be harder to upgrade and configure at the time, recently most routers can be configured
from a web interface

17.13
pros:
   when the result of name resolution is not static, name server can be updated once and for all, while static tables
   on each client must be reconfigured
cons:
   name server without backups is a single point of failure, also instead of a table in secondary storage an entire
   protocol must be designed and deployed on both client and server, name resolution will be several order of

magnitudes slower than table access
each client can maintain their own cache of name resolution, skip the RTT on cache hit

17.14
so the protocol has a clear start point and a change can be contained in a small scope
if there's no top level name servers, name resolution requests would have to be propagated all over the network just
like routing information, take minutes instead of fraction of a second
if there are only top level name servers, any change to any domain name must be committed to the top name servers, the
servers would be flooded by modification requests and cannot reply to requests when the resolution table is
being updated

17.15
pros:
absolute and transparent reliability
cons:
when the upper layer does not need reliability, the additional overhead of acknowledgement, error checking &
retransmission is of no purpose
physical layer protocols put restrictions on mediums, more strict protocol limits the mediums the entire network
stack can traverse

17.16
with dynamic routing packets may arrive out of order, if the order is of importance the packets arrived from network
must be buffered and reordered, by the application or by a transport layer protocol e.g. TCP
when latency must be minimized (e.g. telephone, remote medical operation) virtual routing is more appropriate than
dynamic routing

17.17
// skipped

17.18
UDP is not reliable, packets may be lost or reordered on arrival, while a single byte of error in HTTP documents may
render the entire page malformed
one possible solution is connection multiplexing in HTTP/2, once a TCP connection is established multiple data stream
can be carried by it independently, the connection is closed only when all the streams are terminated

17.19
pros:
    the API is compatible with a centralized system, user programs written for a centralized system can be compiled with
    or dynamically linked to the system library
cons:
    a distributed system is inherently different to a centralized one, with shortcomings and benefits the users are
    unaware of, for instance the user may assume a file in secondary storage can be accessed in milliseconds but in fact
    it's located in a remote host and takes seconds to open

17.20
1. remote access, the file system can be accessed from multiple terminals, the home environment of a user can be bought
    to a remote site
2. file name in DFS may be independent to its location, an access to a file can be served by one of many servers
    holding copies of that file, higher availability and fault tolerance
3. capacity of the file system and be extended transparently by connecting another server machine to the network

17.21
NFS is more suitable, since database accesses are typically random and OpenAFS by default uses a bigger cache block,
remote database in OpenAFS will experience more cache miss on the same amount of cache memory, also cache are only
synchronized to server on file close, database in OpenAFS have to deal with inconsistency and merge conflict more
frequently

17.22
NFS is partially location transparent: a remote directory can be mounted to arbitrary local mount point and appear as a
local directory, but the internal structure of the remote directory is exposed hence this scheme is not location
independent at all
OpenAFS by associating each file with a location-independent file identifier provides both location transparency and
location independency, file names are mapped to identifiers then to physical locations

17.23

// cannot think of a situation in which user requires only location transparency but not location independence

17.24
// thanks solutions manual
failure recovery and RPC as a whole is no longer necessary, the same interface of local procedure call can be reused
across the network
virtual circuit should be preferred as there's no link failure and packets may still arrive out of order in dynamic
switching, once the virtual circuit is established it's solid
if the links are also fast client-side cache can be omitted

17.25
// referring http://pages.cs.wisc.edu/~remzi/OSTEP/dist-afs.pdf
all AFS clients launches a daemon called "Venus process", cache of a file is assumed to be valid until the server
contacts the Venus process of a client about an update by another client or the server itself, when reopening a cached
file the client sends the server a validity check request, if the request timed out because of server failure the cache
is discarded and the client will request a new copy of the file from the server

17.26
local and remote cache serve different purposes, local cache ensures that most access to a remote file can be handled
locally, remote (in memory) cache brings data from secondary storage to main memory so it can be read or written many
order of magnitudes faster by the server

17.27
1. whole-file caching: the entire file is transferred on file open and only synchronized on file close, most file
   operations performed locally to reduce the server load
2. file status callback: the server notifies the clients on file modification, no heart beat packets are transferred
   from clients to server

17.28
// the relative text must be edited out, the only mention to the Apollo Domain is using a timestamp as one part of the
// file name

17.29

whole-file caching vs periodical write back
stateful (callback list) vs stateless

18.1
cons:
1. a subset of these modules must be loaded at startup which means longer launch time than a monolithic kernel
2. a malicious program may be able to escalate its privilege by injecting kernel modules to the system

when the OS works on a fixed hardware it can be compiled into a single binary file, otherwise it should always be split
into modules or the OS cannot utilize new hardware released after its compilation

18.2
// thanks solutions manual
1. threads can be managed completely in user mode (N:1 model), from the viewpoint of the kernel it's a single process
   while the CPU allocation and context switch among threads of a process is scheduled by a user mode library
2. threads can be managed directly by the kernel (1:1 model), e.g. in Linux each thread is a task as well as a
   single-threaded process
3. a hybrid approach (M:N model), software threads are mapped to (usually fewer number of) hardware threads, the
   mapping is controlled by a user mode library while the CPU allocation and context switch is handled by the kernel

in Linux threads are managed by the kernel (model 2), threads of a process can be executed concurrently on multiple
CPUs, which is impossible for software threads since they are transparent to the kernel, by contrast software threads
takes less time to initialize and switch, also when done correctly software threads can still benefit from asynchronous
and non-blocking IO

18.3
pros:
1. calling kernel procedures or accessing kernel data structures won't cause page fault, these operations will have
   a lower and more consistent delay
2. the OS will not be preempted by page fault while updating the kernel data structure, less synchronization

cons:
1. the kernel consumes more than necessary memory as even a kernel procedure that's never used will never be paged

out, user processes will start thrashing earlier
2. the kernel has to manually release pages once occupied by a variable size data structure

18.4
pros:
1. only one copy of shared library has to be stored in secondary storage and main memory
2. dynamically linked library can be updated without recompiling the main program
3. the same source file can be compiled on different platforms given a dynamically linked libraries as a compatible
interface

cons:
1. linking errors are the one of most prevalent and hardest to fix errors during software development and
deployment (e.g. user may update their libraries to a new version that's incompatible with the program), a
single statically linked binary is guaranteed to work
2. for security concern: the shared library may not be what the program is expecting

18.5
networking sockets:
1. the network stack is standardized by RFCs and available on most platforms, the communicating programs don't have
to contain code for each different platform just for communication
2. network sockets are full-duplex, both sides can send and receive data at the same time free from conflict and
data race

shared memory:
1. it is much faster to write to and read from the shared memory pages instead fo going up and down on the network
stack
2. communication by sockets must copy the data from buffer to buffer, while shared memory contains a single copy of
the data

18.6
// thanks solutions manual
1. modern disk hardware contains on-board controllers which optimizes and reorders disk read and write, no need for the
kernel to do these jobs
2. the geometry of modern magnetic disks are complex and varies from manufacturer to manufacturer, almost impossible to
optimize read and write for all of them

18.7

pros:
1. ensured type safety, higher level of abstraction, less human mistakes
2. automatically prepare stack for procedure calls, align structures and allocate local variables
3. much more readable and easier to write

cons:
1. high level languages are cross platform, system specific operations e.g. saving registers to PCB can only be
   implemented in assembly
2. the compiler occasionally will not optimize the code to the most efficient assembly

18.8
when the environment of the new process must be prepared with heavy computation and exclusive read-write access to the
data in the memory of the parent process, by calling fork() then exec() the environment can be prepared concurrent to
the parent process
when the environment of the new process is immediately available vfork() is more appropriate, otherwise modifying about
anything between exec() and vfork() is UB

18.9
file transfer: TCP, each single bit may be significant to a file, the socket must be reliable
periodical test: UDP or raw ICMP packets, TCP makes too much assumptions and an outage is only visible to application
layer after TCP timeout (1-5 seconds)

18.10
// again an exercise seemingly referring removed section
layered design & generic interface

18.11
pros:
  encapsulation, a kernel module cannot access arbitrary code in the kernel
cons:
  the additional effort to maintain the symbol table exported by the kernel and kernel modules

18.12
1. prevent modules from having unprotected concurrent accesses to a hardware
2. to make the job easier for module loading utilities e.g. modprobe & autoprobe

18.13
clone() syscall accepts flags which specifies what are shared between the parent and child process / thread, if no flag

is set the child shares nothing with the parent thus is equivalent to a child process, if file system, memory, open file
table and signal handlers are all shared then the child is equivalent to a thread

18.14
kernel-level threads, Linux do not distinguish between process and thread in regard to scheduling, all Linux threads are
visible to the kernel and scheduled by the kernel on waiting or ready list

18.15
the objects not shared between child and parent processes - open file table, memory pages and signal handler table -
must be copied on creation
while there's no difference between the scheduling of processes and threads as Linux does not distinguish between them,
switching to another process costs much more CPU time than to another thread as the objects must all be loaded from
backup storage

18.16
traditionally UNIX systems allocate a fixed time slice to each process and allow them to run for the time slice no
matter how much processes are there in the system, therefore the more processes running in a system the longer a process
must wait before its turn to come, by contrast CFS, the new scheduling algorithm introduced in Linux 2.6, allocates 1/N
of CPU time to a process when there are N running processes and each process is allowed to run once every target latency
, reduces the response time of an interactive process when the system is crowded

18.17
target latency and granularity
small target latency & small granularity:
    lower latency, more time wasted on context switching
large target latency & large granularity:
    higher latency, processes run for longer before being preempted, more CPU time on actual job

18.18
Linux kernel offers no guarantees about how quickly a process will be scheduled once the process becomes runnable, to
enforce that each process must specify a deadline and the scheduling algorithm must order the runnable processes by

in respect to their deadline and report an error if it's impossible to meet all the deadlines, thus the systems may not
be able to handle the same amount of processes as before

18.19
when a user process requests access to page not backed by a file, e.g. heap allocation

18.20
when a private page is shared between parent and child processes

18.21
pros:

1. code in shared memory is less critical than kernel code, error in shared library will neither crash the system
   nor escalate to fatal security breach
2. by loading shared library on demand memory consumption of kernel is reduced
3. shared library can be extended easily and can be more stable

cons:

1. an additional level of indirection, less efficient compared to a single kernel binary

18.22
pros:

1. after a crash the system can replay or undo operations and restore consistency
2. the OS can commit IO operations much faster, operations in a transaction can be performed asynchronously if the
   driver and controller of the hardware doesn't support it already

cons:

1. another level of indirection

metadata operations benefit more from the journaling system as multiple operations can be performed in batch

18.23
the implementation of file systems are hidden behind the virtual file system interface, the kernel does not have to
change to support new file systems

18.24
1. setuid in Linux allows a process to drop and reacquire its effective UID repeatedly, effective UID of a process can
   be set to real UID and restored to previous value while in UNIX the only operation possible is switching real and
   effective UIDs
2. fsuid and fsgid, a independent UID & GID pair only used on file accesses
3. right passing between programs

18.25
// again an exercise seemingly referring removed section
// thanks solutions manual
1. vast amount of people examine the source code of Linux kernel so vulnerabilities may be found earlier than closed
source projects (not always true, see OpenSSL/heartbleed)
2. people with malicious intent will find the vulnerabilities earlier too
3. patches to vulnerabilities can be deployed faster, everyone can download the new source code and compile instead of
waiting for the official release of a new version

19.1
32/64-bit preemptive multitasking client operating system
1. supporting both Win32 and POSIX API by client-server architecure
2. shipped with a GUI

19.2
1. security: access rights are controlled by ACLs, granted to users or user groups, in additional to a capability
mechanism called integrity level similar to rings in Multics
2. reliability: code is tested with fuzzing (randomly generated input to programs), memory diagnostic at boot time,
collect user statistics for timely patch
3. Win32 and POSIX dual API compatibility
4. performance
5. extensibility
6. portability
7. i18n
8. energy efficiency
9. dynamic device support

19.3
1. the hardware boot loader is loaded from on board ROM, runs POST diagnostics, sets devices to clean state, load
windows bootmgr from the system disk
2. if the system was hibernating winresume restores the state of the system, otherwise bootmgr loads winload
3. winload loads the hardware-abstraction layer, the kernel and device drivers, execution transferred to the kernel
4. kernel initializes itself, starts system process and SMSS initial user process
5. SMSS loads more drivers, initializes paging files, starts session 0 for system-wide background services, execution
transfers to WINLOGIN

6. WINLOGIN logs on a user, starts EXPLORER the GUI process, LSASS the security subsystem, SERVICES the service control
    manager and CSRSS the Win32 environmental subsystem

19.4
1. the hardware abstraction layer (HAL) that hides the difference between chipsets of the same architecure
2. the kernel which handles thread scheduling, low-level processor synchronization, interrupt and exception handling,
    and switching between user and kernel mode
3. the executive that contains object manager, virtual memory manager, cache manager, security reference monitor,
    plug-and-play and power managers, registry and booting

19.5
managing kernel entities, including processes, threads, semaphores, mutexes, files, sections, ports and other interal
IO objects, generating handles when an object is created or opened, checking whether a process has the rights to access
an object, enforce quotas e.g. the maximum amount of memory allocated to a process, maintaining the internal name space
of objects

19.6
1. creating processes and threads with priorities and processor affinities
2. deleting threads
3. support debuggers by providing APIs to suspend and resume threads or to create threads that begin in suspended mode
4. attach a thread to another process
5. thread impersonation

19.7
a mechanism of full-duplex message passing between a server and a client process on the local system, used by local RPCs
and native Windows services, ALPC supports three message passing techniques: copying data, passing data address and
length, and shared memory

19.8
managing file systems, device drivers, network drivers, buffers of IO devices, controlling the cache manager, working
with the VM manager to provide memory mapped file IO, notify processes for IO events by APCs

19.9
Windows supports both P2P and client-server networking, windows implements transport protocols as drivers in the driver
stack, SMB is a protocol for remote IO requests, PPTP is a protocol for communicating between remote-access server
modules and supports encryption

19.10
as a hierarchy of directories, file entries in each directory is stored in a B+ tree

19.11
in NTFS all file system data structure updates are performed inside transactions, a transaction records redo and undo
information before altering the data structures, after a crash transactions are redone and undone so as to ensure the
metadata is restored to a earlier consistent state, but user files are not protected by this scheme

19.12
same to 19.13

19.13
1. reserve and commit virtual memory by VirtualAlloc()
2. map files to virtual memory by CreateFileMapping()
3. allocate heap memory by HeapCreate(), HeapAlloc() and HeapRealloc()
4. allocate thread local global variables by TlsAlloc()

19.14
when a thread's time quantum runs out the scheduler preempts it by a DPC which is a kind of software interrupt, also to
block the handling of interrupts beside those generated by urgent IO devices

19.15
handle is a reference to an object / kernel entity in kernel space, stored in the handle table of the process, a handle
can be acquired by creating or opening an object, by receiving a duplicate from a communicating process, or by
inheriting from the parent process

19.16
on IA-32: memories are divided to pages of size 4KB or 2MB, the upper 2GB are mapped to system code and data structures
and are inaccessible by user code, including self-map (page table of the process mapped to virtual address), hyperspace

(process specific working set information) and session space (drivers shared among processes in a session), the lower
2GB user memory can be allocated by first reserve virtual pages then commit pages to physical memory
to improve performance, page table in virtual memory of each process always occupies the same virtual addresses, write
out pages by LRU policy, and faults in adjacent pages when a page is faulted

19.17
addresses right after the address allocated by malloc() can be marked as no-access, so buffer overflow will result in
an exception instead of being unnoticed

19.18
1. by copying the data from sender to receiver memory, using the port buffer as an intermediate storage
2. by setting up a shared memory section, both sender and receiver can read and write the section
3. by exposing part of the virtual memory of the sender to the receiver
the communicating channel should choose among the methods by data size of messages

19.19
the cache manager
cache in Windows is divided into blocks of 256KB, each can hold a view of a file, each IO request to a file marked as
cacheable and invalid (not in cache) is passed from IO manager to cache manager, which in turn allocates a cache block,
maps a view of the file into that block, then copies the block into user's buffer
the cache manager predicts the access and prefetches blocks into the cache, and by default dirty blocks are written back
every 4-5 seconds

19.20
file entries of a directory is stored in a B+ tree instead of a table

19.21
a process is a program in execution, the unit of work in a system, a kernel object in Windows with its own virtual
memory space and handle table, a process is created by the CreateProcess() syscall and may contain one or more threads
with their own stacks and registers

19.22

1. fiber is a user-mode facility, kernel is unaware of fibers but threads are scheduled by the kernel
2. fibers are cooperatively scheduled, only explicitly yield the thread to other fibers instead of been preempted
3. fibers shares the same thread-environmental block

19.23
1. fiber is a unit of execution defined in the scope of a single thread, while UMS schedules different user threads to
   the same kernel thread
2. fiber is exposed to the user programs while UMS is hidden from users, may only be utilized by system libraries
3. fiber is unsafe to use because they share the same TEB, UMS does not have such problem

19.24
a UT that will normally block on IO requests waiting for its KT can then explicitly yield the execution to another UT or
be scheduled by a kernel mode primary thread, basically why UMS is possible

19.25
// again an exercise seemingly referring removed section

19.26
Windows uses a multilevel page table for each process, beside the top level not all entries in all levels are valid,
a page-directory entry (PDE) will only point to a valid in-memory PTE or PDE table only when the corresponding virtual
addresses in that range are reserved

19.27
with self-map page table itself can be accessed from virtual addresses, can be paged in and out as any other pages in
the memory space, is backed by paging files or a regular file like normal pages

19.28
it won't work, the entire content of memory is stored on disk and brought back when the system leaves hibernation, if
the amount of physical memory shrank or the number of register changed the same set of pages and registers cannot be
restored

19.29
the suspend counter can be used to simulate a semaphore: a thread can create multiple sub threads and be resumed only

when all sub threads have done their job and resumed the parent thread

20.1
as system libraries e.g. FORTRAN compiler and assembler is stored on magnetic tapes and must be manually loaded and
unloaded on demand, jobs that require similar libraries are executed in batch by the operator

20.2
magnetic tapes are inserted between input card reader and CPU, as well as between CPU and card printer as buffers, which
are later replaced by magnetic disks that can be written to by multiple card readers and read from by multiple output
devices to match the speed difference between CPU and IO devices

20.3
the page replacement algorithm on Atlas assumes that programs access memories in loops, it uses the two most recent
reference to a page to predict the next access to that page, if a page is not accessed by the predicted time it's
written out to backing storage, age of a page is not tracked or decremented

20.4
1, 2 then 4

20.5
Mach can replace BSD with any other operating system interface or execute multiple operating system interfaces at the
same time

20.6
features of operating systems on huge computers will eventually be available in operating systems on smaller computers
or even mobile devices
://os-book.com/OS9/practice-exer-dir/
// whenever possible C programming assignment will be done in Rust

1.1
as stated in the text,
    "In general, we have no completely adequate definition of an operating system."
more or less an operating system has to
    1. provide an interface of hardwares to softwares (may not be the case for embedded systems)
    2. run the user applications in a safe manner (again may not be the case for embedded systems)

3. manage hardware resources

1.2
1. when strictly uniform performance is of importance
   e.g. the operating system has to perform exactly the same on different hardwares, maybe for real-time purpose
2. for security reasons
   e.g. the recent Intel CPU exploits which operating systems sacrifices ~15% performance to patch
3. to provide a backward-compatible user interface to softwares
   the operating system have to support legacy softwares and run in a mode designed for hardwares decades ago

all these "waste" are necessary to fulfill their design requirements

1.3
the running time of all the instructions must be deterministic, which means:
   GC is almost impossible: the operating system cannot randomly pause for a milliseconds and collect garbages
   the language in use must have a very strict semantic model, mapping program to instructions in a well-defined manner
   some probabilistic algorithms and data structures cannot be utilized, e.g. hashtable

1.4
should:
   web browser is an absolutely necessary utility which should be provided by an operating system out-of-box
   compiling an entire web browser is both painful and difficult, web browsers are usually distributed in binary
shouldn't:
   it isn't really part of an operating system, web browser definitely doesn't run in kernel mode
   for linux distros most of their users should know how to fetch a web browser with wget or curl

1.5
user applications cannot run kernel mode privileged instructions directly which may potentially interfere the operating
system and other user applications
they may only delegate access to these instructions to the operating system by means of well-defined syscalls
so these syscalls can be analyzed and verified by the operating system in the context of processes

1.6
a. privileged, otherwise a user application can hold the CPU forever
b. not privileged

c. privileged, otherwise user applications can interfere with each other unchecked
d. not privileged, by design
e. privileged, otherwise a user application can deny syscalls from other applications and hold the CPU forever
f. privileged, hardware is managed by the operating system, not directly by the user applications
g. privileged, otherwise renders the whole dual-mode system useless
h. privileged, IO devices are managed by the operating system, user applications should use syscalls

1.7
1. the operating system, loaded into memory by the firmware on startup from secondary storage, cannot be modified by
   the operating system itself, i.e. no update
2. the protection is not complete: an operating system is not entirely static, it has to hold dynamic memories
   for e.g. file system

1.8
ring -3: Intel ME, a separated chip beside Intel CPU which control and monitor the entire computer from beneath
ring -2: processor microcode, under machine code instructions, the real instructions running on the chip
ring -1: hypervisor of the virtual machine environment, manage multiple virtualized operating systems
ring 0: kernel, operating system

1.9
starting from a time set by the user or synced from a remote server:
   the process asks the kernel to wake it a fixed time later by a syscall and go to sleep
   the operating system schedules the process to run after the fixed time
   the process upon wake up and increment the time by the fixed time

1.10
cache is useful due to the fact that:
   they are faster in term of access time compared to the storage on levels below
   by caching multiple small read / write can be performed in batch, possibly extending the lifetime of the storage
   if it can only survive a finite time of read / write operations
cons: caching complicates access to the storage, also cache invalidation is one of the only two hard things in compsci
even if the cache is as big as the storage device, it may not be persistent, the content still have to be written to the
storage device before power off

1.11
in client-server distributed systems hosts either generate or satisfy requests, in peer-to-peer distributed systems
every host do both of them

1.12
a. 1. without proper isolation one user may have access to all the sensitive contents belongs to another user
  2. critical failure of one user process may affect or even terminate other processes
b. virtualization may be able to provide the security, but in practice there's always another exploit yet to be found

1.13
a. CPU, memory, storage devices
b. CPU, memory, IO devices, network links
c. (in addition to workstations) battery

1.14
1. when the user only use no more than a fraction of the hardware capacity, part of a time-sharing system can be
  rented instead of an entire PC or workstation
2. when the user works in a group and have to share read / write access to some files within the group, it's cheaper to
  work on a time-sharing system then doing so via network links

1.15
symmetric:
  all CPU cores are equal in functionality, with the same amount of SRAM cache, computing power and access to the
  main memory, tasks are distributed in software level as well as communication between cores
  common design of desktop CPUs
asymmetric:
  CPUs have different functionality: a main CPU controls how work is distributed among worker CPUs
  software software communicates solely to the main CPU
  common design of mobile CPUs
pros (compared to multiple single-processor PC):
  cheaper, no duplicated IO device and IO bus
  communication / memory sharing among processors are faster by magnitudes
  easier to program: just spawn threads and the operating system can distribute them evenly among processors
cons:

it is a single PC with a single set of IO device at the end, cannot be used by two people at the same time

1.16
instead of multiple processors managed by a single operating system, clustered systems "are composed of two or more
individual systems—or nodes—joined together"
hardware: a network, wired or wireless
software: a distributed operating system, running on hosts and monitors each other

1.17
1. duplicate each write to the two nodes, distribute reads evenly, higher availability, half efficiency
2. distribute write evenly between two nodes, properly guide reads to the correct node (e.g. by DHT)
   full efficiency, a hardware failure permanently wipes data

1.18
a network computer (thin client) passes most of its tasks, which would be computed locally by a traditional PC, to a
remote server assigned to it
by handing over its tasks, network computer consumes less energy thus has longer battery lifetime
if the data is stored in the server as well, users may access their data from different terminals seamlessly

1.19
to inform the CPU about a software / hardware event so it can handle it
traps are software generated interrupts, indicating either an software error (e.g. divide by zero) or a request of
operating system service (syscalls)

1.20
a. CPU sets up buffers, pointers and counters for the DMA controller, which copies a whole block of data into the
memory accordingly, meanwhile the CPU can do something else
b. by a hardware interrupt
c. thanks https://en.wikipedia.org/wiki/Direct_memory_access#Cache_coherency
   the memory transfer is invisible to the CPU before completion, DMA may cause cache incoherency between CPU cache and
   the main memory

1.21

the published instruction sets may be an abstraction over the real instructions running on the hardware (microcode),
hence a dual-mode instruction set can be build on a mode-less hardware
such an architecure may be exploitable to adversaries with physical access to the computer

1.22
to better handle the typical computing tasks: in practice user applications works on both shared and isolated memories

1.23
depends on the memory ordering of read / write accesses, a write operation performed by one processors may or may not
be visible to other processors immediately
refer https://doc.rust-lang.org/stable/std/sync/atomic/enum.Ordering.html

1.24
a. DMA may perform memory write temporarily invisible to CPU and cache
b. different cores may perform independent read / write thus have different cache before synchronization
c. same to multiprocessor

1.25
dual-mode instructions
memory access is privileged instruction, user applications can only perform memory access through syscalls, operating
system maintains memory region of each process, upon receiving a syscall examines if the process is accessing its own
memory or not, if not return the the infamous segmentation fault error to the user application

1.26
a. LAN, all devices are physically reachable by a network administrator
b. WAN
c. WAN, a neighborhood in general is not an organization

1.27
limited power supply
limited computing power
different IO devices (touch screen instead of keyboard & mouse)
higher requirement of stability (phones are rarely turned off)

1.28
higher availability: client-server architecure has a single point of failure which is the server
better scalability: peer-to-peer architecure in nature is more scalable compared to extending the capacity of a server

1.29
file distribution (e.g. bittorrent)
video streaming
network traffic tunneling (e.g. TOR)

1.30
pros:
  more participants, earlier bug detection, more agile response to needs (for developers)
  lower cost (for consumers)
  provides learning material to students
cons:
  bigger attack surface
  copies cannot be sold

2.1
provide an compatible and easy-to-use interface to user softwares and higher layer system components
hide the real implementation of system functionalities which is subject to change

2.2
// section 1.6
create and deletion
suspension and resumption
synchronization
communication
deadlock handling

2.3
// section 1.7
keeping track of which parts of memory are currently being used and who is using them
deciding which processes (or parts of processes) and data to move into and out of memory
allocating and deallocating memory space as needed

2.4
// section 1.8
free space management
storage allocation
disk scheduling

2.5
to provide a programmable, concise and unambiguous user interface to run system and user applications

it's not implemented as a module of the kernel because it doesn't have to, a shell can be built on top of syscalls

2.6
fork() to create a process inheriting the stack of the current one

2.7
enhance modularity of the operating system, part of the operating system can be built upon the kernel instead of in it

2.8
modularity, the operating system can be implemented, reasoned and debugged one part of a time
but layered approach increases the overhead of syscalls

2.9
GUI
  human friendly access to user and system programs
  a user program in Linux land but a system program in Windows and OSX
  in a closed-source OS, the GUI usually is hard-coded into the system
program execution
  OS essential
  user program cannot access the layer under syscall
IO operation
  OS essential
  user program cannot access the layer under syscall
file system manipulation
  organize file in a human readable way
  it must be part of the OS, user program cannot define their own flavor of file system on the one provided by the OS
resource allocation
  OS essential
  kind of doable in user land if the OS exposes lower level syscalls to memory management functions
  earlier version of Rust used a custom memory allocator instead of the OS default

2.10
in that way the OS doesn't have to be loaded into the main memory
those hardwares simply have no enough memory to load the system

2.11
operating systems write their location on the disk to a special boot block on the disk during installation
on startup the boot loader reads these locations then let the user choose one from them

2.12
1. UI and exposed API: load, IO, file system, communication, error detection
2. internal system functions: resource allocation, resource accounting, memory protection

functions in the first category is visible to the user in form of exposed API and syscalls
functions in the second category is not visible but a promise the OS makes to the user

2.13
1. put in a predefined sequence of registers, e.g. first few parameters in Linux ABI
2. pushed to stack, e.g. after a few parameters in Linux ABI
3. stored in a table in the memory

2.14
build the program with debug symbols, let the operating system interrupt the program frequently with a timer, record
where the execution is each time the program is interrupted
without time profiling optimization of a program of decent complexity is nearly impossible

2.15
// section 1.8.1
creating and deleting files
creating and deleting directories
file and directory manipulating primitives
mapping files onto secondary storage (mmap?)
backup to stable storage media

2.16
pros:
  these two concepts share a lot of resemblance
  a common API to both of them simplifies the system design and its interface
cons:
  not all file operations are possible to all devices
  e.g. seek a tty device results in an error
  certain devices are too danger to be manipulated as a file, pipe something to /dev/hda will wipe the boot block

2.17
possible if the default one is implemented in user land, i.e. all it does is call system programs

2.18
message passing: safer with bigger overhead and requires extra memory
shared memory: faster but must be done right carefully or will cause data race / deadlock

2.19

like any other program, the resulting operating system will be more configurable

2.20
device driver and memory management
memory management must be built upon drivers, but drivers may also need access to the memory

2.21
all but the bare minimum functions of the operating system is implemented as separate programs
user program requests services from the kernel
which in turn passes these requests to the corresponding module program
kernel components in microkernel architecure communicates by inter-process communication, much slower than function
calls in the same program

2.22
so these functions can be implemented and compiled separately and dynamically
installation of a new device can be done without recompiling or even restarting the system

2.23
they are both mobile operating system, serving roughly the same set of hardware and market
iOS is closed-source, Android is partially open-source

2.24
https://web.archive.org/web/20080604070631/http://blogs.sun.com/jrose/entry/with_android_and_dalvik_at
main reason listed: GPL of openJDK, power consumption of JVM, efficiency of assembly v.s. Java bytecode

2.25
https://www.semanticscholar.org/paper/An-Overview-of-the-Synthesis-Operating-System-Pu-Massalin/c8f49d5d94e1f9adf608b7c99e1e731c2381d82a
Factoring Invariants method
   if a system service is called frequently with the same parameters, the execution path of the call is cached and
   carried out next time the same service is called
Collapsing Layers method
   lower layer of services can be invoked directly in execution
Executable Data Structures method
   traversal order is stored alongside the data in data structures
faster syscalls in exchange of less flexible operating system architecure

2.26

```
./OS/posix-file-copy
sudo dtrace -n 'syscall:::entry /execname == "posix-file-copy"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 535 probes
^C
  __mac_syscall                                    1
  access                                           1
  bsdthread_register                               1
  exit                                             1
  fcntl                                            1
  getrlimit                                        1
  issetugid                                        1
  shared_region_check_np                           1
  sysctlbyname                                     1
  thread_selfid                                    1
  close                                            2
  csops_audittoken                                 2
  fcntl_nocancel                                   2
  fsgetpath                                        2
  fstat64                                          2
  getentropy                                       2
  getpid                                           2
  proc_info                                        2
  read_nocancel                                    2
  write                                            2
  ioctl                                            3
  read                                             3
  csrctl                                           4
  open                                             4
  mprotect                                        10
  stat64                                          42
```

3.1
PARENT: value = 5
child process inherits a copy of stack of the parent process, modification in the child process will not reflect to the
parent process

3.2
8

3.3
context switching
short-term process scheduling
accounting of process memory and inter-process communication

3.4
a special syscall switches the CPU to the other register set
some register set has to be swapped out to the memory to make room for the new context

3.5
only c. shared memory segments, others are copied

3.6
similar to the design of TCP
=> request
<= process call
   ACK (lost)
=> request (resend)
<= duplicate sequence number, ignored
   ACK
=> ACK received

3.7
retransmission

3.8
short-term scheduling:
   controls which process in the ready queue could run and allocates CPU time to it
   executes once every ~100ms
medium-term scheduling:
   controls which process should be kept in memory
   swaps a process out from memory is it's waiting on an IO event or others
   executes on every major IO event
long-term scheduling:
   controls in which order pooled processes could run
   executes every time a process terminated

3.9
saves the context of the current process to its PCB structure
loads the context of another process from its PCB structure

3.10
// from a fresh WSL debian
init(1) -> init(2)
init(2) -> bash
bash -> ps

3.11

when a process is terminated without waiting on its child processes, its child processes become orphans
the OS assigns init as the new parent of these orphan processes
init calls wait periodically so these orphan processes can be terminated on completion

3.12
16

3.13
in the child process if the call to fork succeeded

3.14
A: 0
B: 2603
C: 2603
D: 2600

3.15
ordinary pipe:
  message passing between parent and child processes
named pipe:
  abstract over the functionality of some program to provide an standard file interface to other programs
  e.g. a pipe that compresses the data passed to it and store them in a file

```
    // https://en.wikipedia.org/wiki/Named_pipe
    mkfifo my_pipe
    gzip -9 -c < my_pipe > out.gz &
```

3.16
at most once:
  an RPC service on an embedded system controlling a light bulb
  the bulb blinks instead of being switched on / off
exactly once:
  a distributed file system
  a write operation may be lost due to connection failure
any use case of UDP is also a use case of unreliable RPC

3.17
CHILD:
  0, -1, -4, -9, -16
PARENT:
  0, 1, 2, 3, 4

3.18

a. synchronous communication:
        if blocks on both ends the messages are passed from producer to consumer directly, no buffer required
        same as other blocking operations wastes cycles
    asynchronous communication:
        easier to use for programmers
        the messages have to be buffered somewhere
b. either the programmer or the kernel has to carry the burden of buffering
c. extra memory copy & occupation v.s. danger of dangling pointers & data race
    OS must be able to mark memory as shared in the latter case
d. fixed-sized messages: easier to implement but harder to use
    variable-sized messages: the opposite

3.19
./OS/process/zombie-child

```
% ps -l
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY
TIME CMD
  501  2037  1916     4006   0  31  0  4297288   2136 -      Ss                0 ttys001    0:00.03
/bin/zsh -l
  501  2279  2037     4006   0  31  5  4268544    740 -      SN                0 ttys001    0:00.05
target/debug/zombie-child
  501  2289  2279     2006   0   0  5        0      0 -      ZN                0 ttys001    0:00.00
(zombie-child)
```

3.20
// skipped, would be a simple wrapper around a hash set in this stage

3.21
./OS/process/collatz-child

3.22
./OS/process/collatz-child-shm
apparently there's no way to list and remove shared memories in OSX which one forget to unlink

3.23 - 3.25
// skipped, entry level application layer programming

3.26
./OS/process/case-reverse-pipe

3.27
./OS/process/file-copy-pipe

Project 1 — UNIX Shell and History Feature
./OS/process/unix-shell

4.1
1. a web server, single-threaded server can only handle one client at a time even though each client only need a tiny
   amount of computation power and is waiting for IO most of the time
2. video compression on a multi-processor computer, a single-threaded program can only ultilize one of many cores,
   even middle-range CPUs today has dozens of cores

4.2
user threads are
   1. directly created by user program or a thread library
   2. managed solely by user space code
where kernel threads are
   1. created by the operating system in a pre-defined model (M:M, M:N or M:1) to meet the request of user threads
   2. managed by kernel space code

4.3
store and load stack, register and thread local storage
in contrast to process context switch, file system, VM space, file descriptors and signal handlers are not switched

4.4
same as above, in general fewer memory is allocated for threads than for processes

4.5
it must, otherwise one thread blocking on IO will block the other threads on the same LWP as well
this kind of blocking and pause is unpredictable and unacceptable in real-time system

4.6
1. when there's only one processor, any CPU bound program will not have better performance with multithreading
2. when the program is IO bound but they all access the same IO device and the IO device can only serve one thread at
   a time in a very slow pace, multithreading will not provide any significantly better performance

4.7
when they are IO bound accessing different IO devices, instead of the sum of the IO response time, only the maximum of

the IO response time would be necessary

4.8
b and c

4.9
when they are CPU bound, on a multi-processor system all cores can be utilized

4.10
if the program is carefully designed so that the error in client script won't corrupt memory shared among threads
the isolation may still be achieved if the signals can be handled locally to the thread

4.11
as stated in the text: multithreaded program in a single processor system

4.12
a. 10 / 7
b. 20 / 11

4.13
a. only task, the same data set is shared
b. both task and data
c. both task and data
d. task but not data, the same data (web pages) are accessed by each thread

4.14
a. one, this task won't benefit from multithreading
b. depends, if the tasks are uniform 4 is enough, otherwise finer granularity is necessary or some CPU would be idle

4.15
a. 6
b. 2, or 4 if threads are duplicated by fork()

4.16
as stated in the text, the former manages data associated to a task (process or thread) separately , storing only
pointers to the data in the PCB, while the former has different data structure for processes and threads

4.17
CHILD: value = 5, global variables is shared (unsafely) among threads
PARENT: value = 0, each process has its own copy of global variables

4.18
a. the full computing power of the system will not be made use of
b and c may not be distinguishable from the view of the program

4.19
1. resource release e.g. closing a file descriptor
2. write to a log file

// about the difference between Pthread API and Rust thread API:
// they are semantically equivalent but:
// 1. threads in Pthread take input by a void pointer on thread spawning, while Rust threads are a closures
// 2. Rust thread can return a Sized value, which can be emulated with Pthread API by allocate a dedicated chunk of
//   memory for the thread to store the return value and later accessed by the main thread
4.20
./OS/process/pid-manager
a lock (mutex) is necessary to prevent data race
not sure if such a lock or variable length vector is available in kernel space

4.21
./OS/process/multi-thread-statistic

4.22
./OS/process/monte-carlo-pi

4.23
// skipped

4.24
./OS/process/prime-numbers

4.25
// skipped, similar exercise in CSAPP

4.26
./OS/process/fibonacci

4.27
// skipped, similar exercise in CSAPP

Project 1

Project 2
// skipped, similar exercise in CLRS

5.1
system clock is implemented by periodical clock interrupts
if interruption is disabled system-wise, the clock interrupts may be delayed, the assumption that these interruption
occurs once a fixed time period no longer holds
either interruptions of the clock thread should not be disabled for too long, or the system time should be synchronized
to a remote server from time to time

5.2
these synchronization primitives have different semantics and diffferent costs
spin lock:
  busily wait by testing and modifying a shared variable atomically
  used when the critical section is short, i.e. syscalls and interrupts are significant overheads in comparison
mutex lock:
  semantically equivalence of spin lock provided by the kernel
  instead of atomic instructions in a busy loop, the threads waiting for the lock is put to sleep and awaken later
  more efficient than spin lock if the critical section is long
semaphore:
  similar to mutex lock, but instead of a binary lock (one thread working, all other threads waiting), (a fixed
  number of) multiple threads can enter the critical section at the same time
  used when a fixed number of resources have to be managed across threads
adaptive mutex lock:
  a hybrid implementation of lock as both a spin lock and a mutex lock
  when the lock is held by a working thread, threads blocking on waiting spin
  when the lock is held by a sleeping thread, threads blocking on waiting is put to sleep too
condition variable:
  a synchronization mechanism defined by two methods: .wait() and .signal()
  x.wait() blocks the current thread until another thread calls x.signal()
  x.signal() awakes a thread or all threads blocking on x.wait()

5.3
waiting by busily trying to acquire a lock in a loop
alternatively a thread can be put to sleep (by a syscall or a kernel mutex lock) and be awaken later by the OS when the
resources become available
it's not always efficient to avoid busy waiting, these syscalls and interrupts have their own costs

5.4
usually the kernel cannot tell the difference between a thread doing meaningful job and a spinning thread
on single-processor system the spinning thread when active will consume 100% of the computing power of the CPU
also for a resource to be released, the spinning thread must be switched out for another thread to make progress
on multi-processor system the spinning thread will only consume 1/N of the computing power where N is the number of
processors in the system
the resource may be released by another thread running on another processor, switching out the spinning thread may be
much more expansive compared to spinning a few hundred instructions

5.5
without atomic operations the memory order may be arbitrary, decrementation of the counter in one thread may not be
visible from another thread in time, two or more threads may acquire the resource guarded by a binary semaphore

5.6
a binary semaphore is semantically equivalent to a mutex lock

5.7
assume the instructions are interleaved in order:

```
LD R0 BAL   LD R1 ACC
ADD R0 A0
          SUB R1 A1
ST R0 BAL
          ST R0 BAL
```

the BAL memory location storing the account balance may have value BAL - A1 or BAL + A0 while it should have value
BAL + A0 - A1
the operations to the same account must be guarded by a mutex lock

5.8
assume turn is initialized to either i or j (otherwise the program causes UB)
assume all operations on variable turn and flag[i] are atomic
mutual exclusion:
  Pi will only enter its critical section when flag[i] && !flag[j] and vice versa
  the two conditions cannot be satisfied at the same time
progress:
  if Pj is absent when Pi is trying to enter the critical section, flag[i] && !flag[j]
  otherwise flag[i] && flag[j]

if turn == i Pi spins, otherwise Pi set flag[i] = false hence Pj can make progress and set turn = i on exit
bounded waiting:
when two threads compete for the resource, only the thread pointed by variable turn can make progress
on exit turn is set to another thread, so a blocking thread at most have to wait for another thread to exit

5.9
// thanks
https://personal.cis.strath.ac.uk/sotirios.terzis/classes/CS.304/Remaining%20Contemplation%20Questions.pdf
mutual exclusion:
the first thread Pi entering critical section has checked that

1. itself has state in_cs
2. all other threads has state != in_cs

any thread entering state in_cs later cannot exit the first while loop
progress:
if multiple threads are in state want_in, all but the one immediately after turn is stuck in a spin lock
bounded waiting:
on exit, threads increments turn by at least 1 (skips idle threads)
a waiting thread cannot be skipped, after at most n - 1 turns this thread will be served

5.10
in single-processor system it's important that a single thread should not hold the CPU too long, which is enforced by
a timer, otherwise a dead lock will freeze the entire system
disabling all interrupts will disable the timer interrupt at well, let a single thread hold CPU indefinitely

5.11
disabling / enabling interrupts to all threads on demand of a single thread is way too expensive

5.12
// thanks https://stackoverflow.com/questions/4752031/
trying to acquire a semaphore may put the thread to sleep, which defeats the purpose of spin lock: spin locks should
only be used if the critical section is short, sleeping with a spin lock will block all other threads trying to acquire
the spin lock for a long time

5.13
linked list:

    if a thread is deleting a node when another thread is trying to insert a node before it, the pointer to the previous
    mode may be dangling, the inserting thread would be dereferencing NULL pointers and cause segfault
bitmap:
    setting the same field to both 0 and 1 in different threads may have varying result depends on the order of their
    instructions

5.14
assign id 1 to n to n threads
each thread acquire the lock by
```
    while (compare_and_swap(&lock, i, 0))
```
and set the lock to the next non-idle thread based on a flag[n] array

5.15
```
void acquire(lock *mutex) {
    compare_and_swap(&lock->available, 0, 1);
}
void release(lock *mutex) {
    test_and_set(&lock->available, 0);
}
```

5.16
instead of while loop, acquire invokes a syscall with the lock (implemented in the kernel), the kernel then put the
thread to sleep and insert it into a waiting list associated to that lock

5.17
short duration: spin lock
long duration: mutex or spin lock, depends on the number of cores available and the time
sleep while lock: mutex by 5.12

5.18
2T

5.19
the second one is much more efficient, the variable hit is never read, its value is of no importance to the service
threads, a mutex lock may put threads to sleep while atomic operations will be compiled to a hardware instruction

5.20

a. all operations on number_of_processes may cause data race, the real number of processes exceed MAX_PROCESSES
b. before each read and write to number_of_process
c. without a mutex lock two threads may call allocate_process() at the same time when
        number_of_processes == MAX_PROCESSES - 1
    the maximum number of processes will be exceeded

5.21
initialize a semaphore to the maximum number of sockets
a thread opening a connection has to acquire the semaphore at first, block if the semaphore is 0

5.22
such a lock may be useful when concurrent quick read is frequent, write is sparse
if write is as frequent as read or the read operating is long, the threads trying to write may be starving

5.23
assume read operation to semaphore is atomic

```
typedef struct {
    boolean block;
    int count;
} semaphore;
void wait(semaphore *lock) {
    while(true) {
        while (test_and_set(&lock->block))
            ;
        // acquired lock
        if (lock->count > 0) {
            lock->count -= 1;
            break;
        }
    }
}
void signal(semaphore *lock) {
    while (test_and_set(&lock->block))
        ;
    lock->count += 1;
}
```

5.24
the child thread must inform the parent thread about a ready number in some means (e.g. condvar), or while very
inefficient the parent thread can check an atomic count of ready numbers in a loop
./OS/process/fibonacci-iterative

5.25
mutual exclusion of a monitor can be emulated by a semaphore guarding all methods of that monitor

5.26
// skipped, straight forward

5.27
// thanks solutions manual
a. if the content is lengthy, copying it to / from the buffer may take considerable time, another thread trying to
    read or write the buffer must wait in between
b. instead of produced buffer elements, the buffer may store pointers to these elements, copying these pointers would
    be far more efficient
    threads must be careful that these memory region should not be accessed once its put into the shared buffer

5.28
in a common use case of read-write lock, there will be much more read threads than write threads
if the lock does not favor write threads, a write attempt will block until there's no thread reading the resource
such an event may simply never happen if the resource is read frequently
a possible solution is, if there are write threads waiting on the lock, all incoming read threads are blocked
in this case write threads are will starve but read thread may

5.29
calling signal() on a conditional variable awakes one thread blocking on the corresponding wait() call, it there's no
threads blocking on the conditional variable the call to signal() has no effect
in contrast calling signal() on a semaphore will always release one allowance to the poll

5.30
threads calling signal() no longer has to wait the awakened thread to exit
all operations on the next semaphore and next_count variable can be eliminated

5.31
along with the resources and conditional variable, the monitor maintains a waiting priority queue
when no printer is available, incoming threads enter the waiting queue and wait on the conditional variable

whenever a printer is released, all threads waiting on the conditional variable are awaken (pthread_cond_broadcast)
they in turn check if they are the top of the priority queue and there's printer available, if so they acquire a
printer, otherwise they call wait() again

5.32
with Rust Condvar and Mutex:
```
struct FileSharing {
    sum_lock: Mutex<u32>,
    cv: Condvar,
    limit: u32,
}
impl FileSharing {
    fn new(limit: u32) -> Arc<Self> {
        Arc::new(Self {
            sum_lock: Mutex::new(0),
            cv: Condvar::new(),
            limit,
        })
    }
    fn acquire(&self, num: u32) {
        let mut sum = self.sum_lock.lock().unwrap();
        while *sum + num >= self.limit {
            sum = self.cv.wait(sum).unwrap();
        }
        *sum += num;
        // acquired file, enter critical section somewhere else
    }
    fn release(&self, num: u32) {
        let mut sum = self.sum_lock.lock().unwrap();
        *sum -= num;
        drop(sum);
        self.cv.notify_all();
    }
}
```

5.33
upon release the calling thread is no longer holding shared resources, so it makes more sense to continue execution in
this case, if the control is transferred to the notified thread the calling thread may have to call release() in
async manner, e.g. spawn another thread to call it

5.34

a.
```
monitor rw_lock {
      int read_count;
      int write_count;
      initialize() {
         read_count = 0;
         write_count = 0;
      }
      read() {
         await(write_count == 0);
         read_count += 1;
         // critical section
         read_count -= 1;
      }
      write() {
         await(read_count == 0 && write_count == 0);
         write_count += 1;
         // critical section
         write_count -= 1;
      }
   }
```
b. there's no mechanism there to notify the threads and the await() function that the value of the expression has
   changed, the expression has to be evaluated from time to time even if the expression is expensive
   also the system instead of the user program has to decide which thread to awake first when the condition is
   satisfied, without specific knowledge of the user program the decision made by the system may not be efficient

c. the paper in mention is paywalled
   maybe the expression has to be simple boolean expressions of local variables so its value can be tracked

5.35

with Rust Condvar and Mutex:
```
struct AlarmClock {
   clock_lock: Mutex<u32>,
   cv: Condvar,
}
impl AlarmClock {
   fn new() -> Arc<Self> {
      Arc::new(Self {
         clock_lock: Mutex::new(0),
         cv: Condvar::new(),
        })
    }
    fn wait(&self, ticks: u32) {
        let mut clock = self.clock_lock.lock().unwrap();
        let now = *clock;
        while clock.wrapping_sub(now) < ticks {
            clock = self.cv.wait(clock).unwrap();
        }
    }
    fn tick(&self) {
        let mut clock = self.clock_lock.lock().unwrap();
        *clock = clock.wrapping_add(1);
        self.cv.notify_all();
    }
}
```

5.36
// already done in 4.20
./OS/process/pid-manager

5.37
a. available_resources
b. two call to either decrease_count() or increase_count() may cause inconsistent write to the variable
c. ./OS/process/resource-counter

5.38
./OS/process/resource-counter

5.39
./OS/process/monte-carlo-multi

5.40
// skipped, no OpenMP in Rust

5.41
./OS/process/barrier-point

Programming Project 1
./OS/process/sleeping-ta

Programming Project 2
./OS/process/dinning-philosophers

Programming Project 3
// skipped, simply a VecDeque in a Mutex

6.1
n! if the system is non-preemptive
(Σti / q)! if the the system is preemptive, length of ith process is ti, the time quantum is q

6.2
non-preemptive scheduling will not pause a process and switch context unless it exits / waits on resources by a syscall
preemptive scheduling may pause a process midway to ensure fair share or to start a process of higher priority

6.3
a. P1: 0 -> 8
   P2: 8 -> 12
   P3: 12 -> 13
   avg turnaround: 10.533
b. P1: 0 -> 8
   P3: 8 -> 9
   P2: 9 -> 13
   avg turnaround: 9.533
c. P3: 1 -> 2
   P2: 2 -> 6
   P1: 6 -> 14
   avg turnaround: 6.866

6.4
processes with high priority are usually interactive, a shorter time quantum will let it be executed more frequently
hence reduce the response latency
processes with low priority are usually CPU bound daemons doing heavy computing job, a longer time quantum will reduce
the time wasted on context switching

6.5
a. SJF is a special form priority scheduling, priority is decided solely by burst time
b. FCFS is a special feedback queue with a single queue running FCFS algorithm
c. FCFS is a special priority scheduling where all processes are assigned equal priority and ties are broken by FCFS
d. different, RR is preemptive and SJF is non-preemptive

6.6

a process with low priority (e.g. a CPU-bound processes) will eventually be assigned higher priority since it's never
executed in the recent past due to its low priority

6.7
on systems with M:1 or M:N thread model, the thread library uses the PCS scheme to map user threads to LWP where threads
in a process compete for LWP, then the OS uses SCS scheme to map LWP to a physical CPU where LWPs system-wide compete
for the hardware
switch to SCS scheme on an M:1 or M:N system effectively turns its thread model to 1:1

6.8
same to 4.5

6.9
P1: 80
P2: 69
P3: 65
lower

6.10
the same to 6.4: IO-bound programs should have lower response time and CPU-bound programs should be preempted less

6.11
a. lower response time can be enforced by shorter time quantum and preemptive system, but these changes cause more
    context switch and lower CPU utilization
b. as illustrated in 6.3, to minimize average turnaround time the OS should wait for a short period when there's only
    a few ready processes, which of course increases the average waiting time
c. better IO utilization means more syscalls, more interruption and more context switch

6.12
more than one lottery tickets can be assigned to a process, increasing its winning probability accordingly

6.13
1. pros:
        no shared memory, queues can be safely manipulated by the core at any time
    cons:
        without some other inter-core scheduling mechanism, the processes have to wait in the queue of a busy core even

    if another core is idle, result in worse load balance
2. pros:
    load balance is naturally assured, all idle cores will get processes from the global queue
  cons:
    the queue must be protected by synchronization tools, access to the queue may be unsafe or at least take
    longer time

6.14
a. the prediction is always 100ms
b. the CPU burst history has almost no effect on the prediction

6.15
CPU bound, a process which do not invoke a syscall for IO in its time quantum will have a higher priority and longer
time quantum next time it's executed

6.16
a. FCFS:
    0 -> 2: P1
    2 -> 3: P2
    3 -> 11: P3
    11 -> 15: P4
    15 -> 20: P5
  SJF:
    0 -> 1: P2
    1 -> 3: P1
    3 -> 7: P4
    7 -> 12: P5
    12 -> 20: P3
  priority:
    0 -> 8: P3
    8 -> 13: P5
    13 -> 15: P1
    15 -> 19: P4
    19 -> 20: P2
  RR (time quantum in order):
    P1, P2 (t = 1), P3, P4, P5, P3, P4, P5, P3, P5 (t = 1), P3
b. see part a
c. turnaround time - burst time
  result skipped

6.17
// skipped, unclear how the priority should affect the RR schedule

6.18
if all the user threads are assigned a 0 nice value, decreasing the nice value of a process without proper permission
may be considered as privilege escalation

6.19
b and d, a if the system is non-preemptive and the first coming process does not halt

6.20
a. the process is executed twice as frequent as other processes
b. alongside the pointers store a time quantum specific to that process

6.21
a. one round consists of: 11 CPU burst of length 1 and 11 context switch
   11 + 0.1 * 11 = 12.1 ms in total, CPU utilization is 11 / 12.1 = 0.909
b. one round consists of: 10 CPU burst of length 1, 1 CPU burst of length 10, 11 context switch
   10 + 10 + 0.1 * 11 = 21.1 ms in total, CPU utilization is 20 / 21.1 = 0.948

6.22
assign higher on priority to their process on creation, invoke dummy syscalls to limit the length of CPU burst so the
processes remains in the higher priority queue

6.23
a. the running process will have highest priority than all waiting processes, the earlier processes in the ready queue
   have higher priority than later processes, effectively FCFS
b. the running process will be preempted by a new coming process, without new processes the running process cannot be
   preempted by waiting processes, new processes have the highest priority, a preemptive LIFO queue

6.24
a. no discrimination at all
b. no discrimination at all
c. short processes will be assigned higher priority by their short CPU burst

6.25
// skipped, refer table 6.22

6.26
HIGH_PRIORITY_CLASS and HIGHEST

6.27
// skipped, refer table 6.23

6.28
// thanks https://en.wikipedia.org/wiki/Completely_Fair_Scheduler#Algorithm
both CPU bound:
  vruntime of A will decay faster than B
A IO, B CPU:
  A will have a slower increasing value of vruntime which also decays faster
  whenever A is ready B will be preempted after its time quantum
A CPU, B IO:
  A will have a faster increasing value of vruntime which decays faster
  potentially both A and B will have vruntime float around a limit, which depend on their burst time and nice value
  which ever have a lower stable vruntime value has higher priority

6.29
as stated in the text, when a higher priority process requires use of a shared resource currently locked by a lower
priority process, the lower priority process temporarily inherits the higher priority to prevent being preempted by
another process
a possible solution is to transfer the shares of the blocking high priority process to the locking lower priority
process so the lock can be released eariler

6.30
priority of a process in rate-monotonic scheduling is fixed, when there are a huge number of processes CPU utilization
of rate-monotonic scheduling is limited to ~70% but EDF may be able to fully utilize the CPU

6.31
a. P1 has higher priority
  0 -> 25: P1
  25 -> 50: P2 (5 remaining)
  50 -> 75: P1
  P2 miss deadline, cannot be scheduled by rate-monotonic scheduling
b. 0 -> 25: P1 (d = 50)
  25 -> 55: P2 (d = 75)
  55 -> 80: P1 (d = 100)
  80 -> 110: P2 (d = 150)
  110 -> 135: P1 (d = 150)

6.32

they may affect both response and turnaround time

7.1
1. a traffic jam in which cars from both sides waiting for the other to yield
2. a quarrel between two children, both waiting for the other to apologize
3. a draw by threefold repetition in chess, both players waiting for the other to make a mistake

7.2
when processes do not require and release all the needed resources at once, i.e. the resources locked by a process can
be partially released so another process can make progress

7.3
./OS/process/banker

```
a.  0   0   0   0
    0   7   5   0
    1   0   0   2
    0   0   2   0
    0   6   4   2
```

b. is safe
c. can be granted immediately

7.4
there will be no hold and wait situation in this system: any process trying to acquire any resource must first acquire
F, but F is mutual exclusive thus only one process can acquire F, a process either hold F or no resource at all, but the
resource utilization is much lower than circular-wait scheme in which resources can be acquired by multiple processes

7.5
each iteration of step 2 and 3 takes O(mn) time, each iteration marks one finish[i] as true, O(n) iterations in total
O(mn^2) operations in total

7.6
a. if the deadlocks can be completely avoided, the system will save 20 dollar per month
b. the turnaround time is increased by 20%, if the jobs are time critical that would be unacceptable

7.7
if the synchronization tools used by the process is managed by the system (mutex, semaphore, about anything except a

spin lock), the system can scan periodically over the waiting list of a mutex and detect that a process is never run
for a long time

7.8
a. no, the processes now can be preempted
b. a process may be constantly preempted by incoming new processes, never gain access to needed resources

7.9
when the state is unsafe, no process can make progress without another thread releasing its resources first, by
definition the state is a deadlock

7.10
impossible, the hold and wait condition cannot be satisfied

7.11
mutual exclusive: each intersection can only be occupied by traffics from one direction at a time
hold and wait: all the four line traffics are occupying an intersection and waiting another to be empty
no preemption: traffics cannot be removed from the intersection without passing it
circular wait: illustrated by the figure

7.12
possible, write attempts are still mutual exclusive

7.13
if only the process running do_work_one is preempted by process running do_work_two after
   pthread_mutex_lock(&first_mutex);
for e.g. fairness then the deadlock will occur

7.14
assign a unique total ordered id to the accounts, always lock the account with a smaller id first

7.15
a. circular-wait scheme incurs less overhead than banker's algorithm, no quadratic or higher complexity procedure is
   running periodically, the requests can be fulfilled or rejected in O(1) time
b. throughput of circular-wait scheme is considerably lower than banker's algorithm as some rejections in circular-wait
   scheme are totally artificial, indicating no potential unsafe state or deadlock

7.16

a. safe, ignore these new resources and the sequence is still safe
b. unsafe, after a process releasing its resources the next process in the sequence may not be able to acquire needed
    resources
c. unsafe, same to b
d. safe, the same set of resources can be allocated to the process anyway
e. safe as long as they request resources through the deadlock avoidance system
f. safe

7.17
by pigeon hole principle there will be at least one process being allocated two resources and can accomplish its job

7.18
for a deadlock each process must be holding less than necessary resources and there should be no resources available
the n processes can only be holding less than m + n - n = m resources, there should be resources available

7.19
use banker's algorithm and set max resource need to 2 for each philosopher
or simply if it's the first request by a philosopher, there should be 2 chopsticks available

7.20
assume the philosopher holding most chopsticks is holding n, there should be 3 - n available in the system

7.21
let available = [1, 1], max = [[1, 1], [1, 1]], no request would be rejected but if
    P0 hold A and request B
    P1 hold B and request A
there may be a deadlock

7.22
./OS/process/banker
a. unsafe, no safe sequence
b. safe

7.23
./OS/process/banker
a. (P0, P3, P4, P1, P2)
b. can be granted
c. cannot be granted, unsafe state and possible deadlock

7.24
processes that acquire resources will eventually releases these resources
when the programmer is careless a resource may not be released, the system may not be able to recover these resources
automatically on process exit

7.25
a state in mutex and a condvar
```
enum Bridge {
    South(u32), // south villager increment counter, north villager block on condvar
    Empty,      // both pass and change to state above or below
    North(u32), // south villager block on condvar, north villager increment counter
}
```
when a passenger reached the other end the counter is decremented by 1

7.26
the state additionally contains a traffic light which switches between two colors periodically (modified by a separate
thread or implicitly inferred from the system time by villager threads)
when the light is GREEN only south villagers can pass
when the light is RED only north villagers can pass

7.27
./OS/process/vermont-bridge

Programming Project
./OS/process/banker

8.1
1. logical address has a fixed range defined by the architecure (2^32 in x86, 2^48 in x64), while physical address has
   range varying from individual system to system according to the hardware installed
2. logical addresses can be non continuous due to segmentation and paging, but physical addresses are always linear
   and continuous

8.2
pros:
   code section cannot be modified, the same program can be loaded into the memory once and executed by several
   processes without race condition
cons:
   cannot think of anything, the PCB is slightly bigger

8.3
so a logical address can be easily partitioned to a page index and a page offset
if the page size is 2^n, the lower n bits of the logical address will be the page offset

8.4
assume N bit words
a. 16 + N
b. 15 + N

8.5
the two blocks of logical addresses will point to the same block of physical addresses, from the view of the program if
the two blocks are read-only they appear to be two blocks with the exactly same content
copying a block of content can be implemented as changing the entry of the target logical address in the page table to
the source physical address, but the two blocks must be read-only from now on or write to one address will be visible
from another

8.6
a segment table
the standard C library may be loaded to the memory once and multiple processes may have different base-limit pair in
their segment table pointing to the same physical memory of standard C library

8.7
a. a segment table loaded to a specific register and searched by the CPU
b. a page table loaded to a specific register and searched by the CPU

8.8
a. all memory access are directly performed to physical addresses with key 0
b. the only user should have root privilege, logical memory addresses can be accessed with key 0
c. if a block of memory is assigned to a process, along with its entry in page table the OS should also store the key
   so that only the process can access the memory, unauthorized access will be trapped to the CPU
d. no difference to part c
e. key is stored beside page table entry
f. key is stored beside segment table entry

8.9
internal fragmentation is inherent to a partition system, e.g. memory in the system can only be allocated a fixed-size

block at at time so any allocation not a multiple of the block size will contain a portion of memory that's occupied
from the perspective of the OS but not usable by the user program
external fragmentation is caused by the variant size of allocations in a system, rapid load and unload of processes may
break the free memory space into small pieces, which has enough combined size to house a large allocation of a new
process but none of them alone is large enough

8.10
// refer chapter 7 of CSAPP
two main tasks are symbol resolution and relocation
when generating relocatable object files compiler should also generate symbol tables so the linker can resolve each
reference to exactly one symbol
then the same type of sections in the object files (read-only data, global variable, etc.) are merged into one, symbol
references are relocated to correct run-time addresses directed by the relocation entries generated by the assembler

8.11
first fit:
    185, 100, 150, 200, 17, 125
best fit:
    300, 100, 350, 17, 10
worst fit:
    300, 600, 350, 200, 750, 125
    300, 600, 350, 200, 635, 125
    300, 600, 350, 200, 135, 125
    300, 242, 350, 200, 135, 125
    300, 242, 150, 200, 135, 125
    (cannot allocate 375)

8.12
a. each process is assigned a continuous block of memory on creation, processes may have to claim the amount of memory
    they want before creation
    heap allocations then can be direct or managed by an allocator to minimize external fragmentation
    if memory locations cannot be relocated in run-time, dynamic memory allocation beyond a certain limit is impossible,
    processes cannot be moved to a bigger continuous block of memory on fly
b. the OS must allow processes to create and delete segments dynamically
c. no special treatment is required

8.13
a. allocations in contiguous memory allocation and pure segmentation scheme all ultimately have to be managed by an
    allocator, and the severeness of external fragmentation is determined by the strategy used by the allocator. pure
    paging does not suffer from external fragmentation as memories are always allocated as a fixed-size block, and
    continuous logical memory does not have to be mapped to continuous physical memory
b. contiguous memory and pure segmentation do not incur internal fragmentation, pure paging cannot allocate memory not
    of size multiple to the page size hence each allocation causes on average half a page to be wasted
c. impossible in continuous memory, straight forward in pure paging and pure segmentation

8.14
if the process does not own a page, the page identifier is not contained in its page table, referring that page will
result in a page fault and be trapped to the OS
logical page in page tables of processes may be mapped to the same frame in physical memory, but the shared memory in
most case should be read only

8.15
backing storage, usually flash memory, in mobile operating systems are not particularly bigger than the main memory,
swapping will not significantly enhance multiprogramming and may damage the flash memory which only tolerate very
limited number of write compared to magnetic disks

8.16
following 8.15, the secondary disk is usually more spacious than the built-in memory, and damaging it would not affect
the memory storing the operating system

8.17
segmentation uses about 2 usize (16 bytes on x64) for each segment id, paging without hierarchical structure for quick
access must contain an entry for each page id, possibly occupies megabytes per process

8.18
first it provides another layer of safety to memory accesses: unmatched ASID is treated the same way as TLB miss, also

ASID allows a single TLB to store virtual memory mapping for several processes, so the multiple level of TLBs do not
have to be flushed after a context switch

8.19
// thanks solution manual
a. if the process cannot be relocated on fly, the amount of stack space must be pre-determined on process creation
    the amount of stack space must be big enough to house the deepest call, so most of the time a large portion of the
    stack space is unused
b. the stack segment still have to be big enough to house the deepest calls, but it does not have to be continuous to
    the .text and .data segments
c. pure paging can handle this scheme with ease

8.20
a. (3, 13)
b. (41, 111)
c. (210, 161)
d. (634, 784)
e. (1953, 129)

8.21
a. 2^(21 - 11) = 1024
b. 2^16 = 65536

8.22
unlimited

8.23
a. 8 + 12 = 20
b. 6 + 12 = 18

8.24
// errata: http://os-book.com/OS9/errata-dir/os9c-errata.pdf
a. 2^(32 - 12) = 1048576
b. depends on the physical memory installed on the system

8.25
a. 100ns
b. 75% * (100 + 2) + 25% * 200 = 126.5

8.26

// thanks
https://faculty.psau.edu.sa/filedownload/doc-6-pdf-214a793090ffcaee487d7c0e1d5d23b0-original.pdf
paging can totally eliminate external fragmentation which pure segmentation suffers, and a long and sparse page table
can be segmented to reduce the memory occupation

8.27
with segmentation the single base-limit pair can be shared among processes, in pure paging each frame of the reentrant
module have to be mapped to a page in each process sharing that module

8.28
a. 649
b. 2310
c. segfault
d. 1727
e. segfault

8.29
when the logical address space is huge and sparse, a single level page table may be enormous, occupies unacceptable
amount of memory, by segment the page entries, lots of the entries in the top level page table can be left empty, pages
not used by the process won't have their mapping to physical memory stored in the table

8.30
if location of the four page tables are stored in registers, each memory load first loads the entry of the right page
table, then the frame location stored in that entry, two operations in total

8.31
if the segmented, hierarchical page table scheme has n level, a memory load is translated to n + 1 memory operations
when the load factor p is above 1, linked list in a single level hashed page table have average length p, a successful
access has to traverse p / 2 nodes on average
when p < 2n, hashed page table is faster, otherwise hierarchical page table is faster

8.32
a. assume 4K page size
   from the 48-bit logical address <s: 13, g: 1, p: 2, offset: 32>:
      g determines the descriptor table (local or global) to be searched

the segment pair (base limit) corresponding to the segment number s in that descriptor table is extracted
CPU checks whether the process has the privilege to access that segment with protection field p
the 32-bit offset is compared to the segment limit and added to the segment base, producing a 32-bit linear address
from the 32-bit linear address <p1: 10, p2: 10, d: 12>:
the top level page table is accessed with p1, returns the location of the second level page table
the second level page table is accessed with p2, returns the location of the physical frame
d is added to the base of the physical frame, producing a 32-bit physical address
b. segmentation + hierarchical page tables, 8.26 and 8.29 combined
c. accessible physical memory is limited to 32 bit or 4GB, but the main reason IA-32 is not adopted by every single
manufacturer is competition for the sake of possible market domination from IBM etc.

8.33
// skipped, why is this even a programming assignment

9.1
a page of virtual memory is not in physical memory when the process is trying to access it
1. the page fault is trapped to the OS
2. the process is switched out, its registers and states are saved to its PCB
3. if the access is illegal (e.g. out of bound), a signal is thrown to the process and most of the time the process is
terminated on next execution
4. assign a free page if any to the process, call the backing store device to read the target frame to the page, put
the process in the waiting queue of that device
if no or few free pages are available in the system some processes are suspended, all of their pages are freed
5. the CPU is allocated to other processes in ready queue before the read is complete
6. the IO device interrupt the CPU, the currently running process is paused and saved to PCB
7. the page table is corrected, the page caused the page fault is now in memory
8. once the process is allocated CPU again, it continues execution from the instruction which caused page fault

9.2
a. n, each page must be brought into physical memory on first use
exactly n page faults occur when the first n references are to each page once, and all following references are to
the same page
b. if m >= n, all n pages can reside in the physical memory, n page fault at most
if m < n, with FIFO replacement algorithm each reference may cause a page fault, p at most

9.3
9EF -> 0EF
111 -> 211
700 -> not in memory
0FF -> not in memory

9.4
a. 4, no anomaly
b. 1, anomaly
c. 5, no anomaly
d. 3, anomaly

9.5
1. the ability to translate logical memory to physical memory through a page table and possibly multiple level of TLB
2. a large secondary storage to store swapped out processes and pages
3. the ability to restart the execution of an instruction after a page fault so that the memory is not left at some
    inconsistent state

9.6
seeking time on average is 60/3000/2 = 10 millisecond
transfer time of a page is 10^3 / 10^6 = 1 millisecond
99% instructions do not access another page, 1 microsecond
1% * 80% = 0.8% instructions access another page in memory, 2 microsecond
1% * 20% * 50% = 0.1% instructions access another page not in memory and the replaced page is clean
    the page has to be sought and transferred to memory, 11002 microsecond ignoring time spent on interrupt handling
1% * 20% * 50% = 0.1% instructions access another page not in memory and the replaced page is dirty
    the page has to be written to the dram first, 22002 microsecond
34.01 microseconds on average

9.7
assume the page size is measured in int (word), matrix is stored in row major
a. each 4 iterations reference two new pages, the process is in constant using and will never be replaced
    100 * 100 / 4 = 2500 page faults
b. each 400 iterations reference two new pages
    100 * 100 / 400 = 25 page faults

9.8

```
./OS/memory/replacement
1 physical frames
   FIFO: 20 page faults
   LRU: 20 page faults
   OPT: 20 page faults
2 physical frames
   FIFO: 18 page faults
   LRU: 18 page faults
   OPT: 15 page faults
3 physical frames
   FIFO: 16 page faults
   LRU: 15 page faults
   OPT: 11 page faults
4 physical frames
   FIFO: 14 page faults
   LRU: 10 page faults
   OPT: 8 page faults
5 physical frames
   FIFO: 10 page faults
   LRU: 8 page faults
   OPT: 7 page faults
6 physical frames
   FIFO: 10 page faults
   LRU: 7 page faults
   OPT: 7 page faults
7 physical frames
   FIFO: 7 page faults
   LRU: 7 page faults
   OPT: 7 page faults
```

9.9
// thanks solutions manual
the valid / invalid bit can be used as a reference bit
when the reference bit has to be cleaned e.g. in second chance algorithm, the valid bit is set to invalid instead, next
access to the page will be trapped to the OS even the page is in physical memory, OS then set the bit to valid
under this scheme a memory access would be trapped to the OS even if the page is in physical memory, dramatically
increasing the cost of memory load

9.10
it's not optimal

if a sequence of references can be scheduled with k page faults with n physical frames, the same schedule can be applied
to any system with n + p physical frames by simply ignoring the additional p frames, hence a page-replacement algorithm
that experiences Belady's anomaly cannot be optimal

9.11
FIFO:
    segments are allocated physical memory one beside another, when segments must be replaced for a new allocation, the
    old segments are swapped out one by one from an most_early_segment index until the consecutive free memory is large
    enough to house the new segment, the physical memory is managed must like the circular buffer in section 5.7.1
    compaction is performed from time to time if the segments can be rearranged
LRU:
    // thanks solutions manual
    select the oldest segment among the large enough ones (including the holes on two sides of them), if there's no
    large enough segment a chunk of oldest neighboring segments by some criteria are swapped out
    if the segments can be rearranged the segments can be compacted and sorted by last access time, the first
    replacement after compaction is exactly the same to FIFO

9.12
a. either the processes are thrashing or all of them are IO bound, without additional information e.g. page fault rate
    it's unsafe to increase the level of multiprogramming of the system, if processes are thrashing the paging system
    may be improved so that processes are given enough pages or swapped out
b. some CPU bound process are busy doing its job, CPU utilization cannot be improved any further by increasing the
    level of multiprogramming
c. processes are either idle or waiting for some non IO events, increasing the level of multiprogramming is a valid
    option to improve CPU utilization

9.13
it's impossible if the base and limit are not on a page boundary, normally a page table cannot limit accesses to only
part of a page

9.14

TLB miss, no page fault:
    the page is loaded but replaced by the TLB replacement algorithm later
TLB miss, page fault:
    the page is not loaded into physical memory and on backing store
TLB hit, no page fault:
    an ideal scenario, quick memory load
TLB hit, page fault:
    incoherent cache, should be impossible, when a page is replaced its entry in TLB must be flushed, the system or the
    hardware has a bug

9.15
a. to Blocked state, the page causing fault must be loaded into memory by an IO device and the process has to wait
b. no change, TLB is hardware and should be transparent even to the OS
c. no change, a success memory load typically does not go through OS

9.16
a. with pure demand paging the only way to brought a page into physical memory is through page fault, all memory
    accesses to the program and the first locality will cause page fault, page fault rate would be around 100%
b. the process during execution moves from one locality to another, when it first enters a new locality it has to
    bring the pages of the locality to physical memory, subsequent accesses in the locality will not cause page fault
c. 1. more pages can be assigned to this process (may be handled automatically after the OS observed a higher than
        threshold page fault rate) taken from other processes
    2. no action is taken, this process will fault from time to time during execution
    3. there's no free memory in the entire system and this process is chosen to be suspended, it's swapped out to
        backing store and its pages are reassigned to other processes

9.17
when a page is marked as copy-on-write, copying this page by e.g. fork() syscall will not duplicate it in the physical
memory but only create an entry in the page table pointing to the same frame, only when the process tries to write to
the page the associating frame on physical memory will be duplicated to a new free frame
with copy-on-write a fork() syscall does not have to immediately copy the entire memory space of the parent process for
the child process, if the child process calls exec() syscall right after fork() the memory copy would be unnecessary

the CPU must be able to understand the copy-on-write bit in the page table so while the attempt to write to a copy on
write page will be trapped to the OS, read attempts will bypass OS and be handled as usual, otherwise the memory access
latency would be unacceptable

9.18
page id: 2715, page offset: 2816
the MMU searches the TLB by the page id, if it found an entry of physical frame location the frame location is combined
with the page offset to produce a physical address, otherwise it start searching through the in memory page table whose
location is stored in a special register
if the entry is found in the page table is found the load is performed as above, otherwise a page fault is trapped to
the OS and the OS will try load the page from the backing store and update the page table, whose content is later loaded
into the TLB by the hardware

9.19
100(1 - p) + (0.3 * 8 * 10^6 + 0.7 * 20 * 10^6)p <= 200
(16400000 - 100)p <= 100
p < 1/163999 = 0.00000610

9.20
since each user thread is mapped to a kernel thread, one thread blocking on IO will not affect the other threads, but
the page table is shared among these threads and should be protected somehow or there's possible race condition

9.21
./OS/memory/replacement
3 physical frames
   FIFO: 17 page faults
   LRU: 18 page faults
   OPT: 13 page faults

9.22
a. E12C -> 312C
   3A9D -> AA9D
   A9D9 -> 59D9
   7001 -> F001
   ACA1 -> 5CA1
b. 4000

c. anything but page 3, 7, A and E

9.23
a. the system is running out of free frames, its trying to reset the reference bit faster to allow more pages to be
    replaced when necessary
b. the system has plentiful of free frames

9.24
when a page is accessed periodically but infrequently, LRU will keep it in the physical memory but LFU will evict it,
possibly make rooms for more frequently accessed pages hence increase hit ratio
when a page experienced a burst of access and is never going to be accessed, LRU will quickly evict it but LRU will keep
it in the physical memory until its access count decays

9.25
when the program initializes a large set of pages, do something else and only then works on the pages, LRU will evict
these pages from the memory so when the process works on them they have to be brought into memory by page faults, in
contrary MFU possibly will keep them in the memory
in almost all other cases LRU are better than MFU

9.26
a. a victim page is chosen from resident pages by FIFO, while the victim page is written out to the backing store a
    free frame is chosen from the free frame pool by LRU, the desired content is read to the chosen free frame then
    the page table is updated, when the resident page is written out it's frame is added to the free frame pool
b. the same frame is reused directly from the free frame pool, the page table is updated without any IO operation
c. a single user system, every new allocation will cause the current loaded page to be written out, the system is
    unlikely to be functional at all
d. FIFO replacement algorithm

9.27
a. no help, the processes are thrashing, a faster CPU cannot manage memory any better than the old one
b. no help, as long as the system has enough disk space to swap out the pages additional disk space has no effect on
    thrashing

c. more processes will be there demanding memory, the state of thrashing would be even worse
d. may help, by decreasing the number of processes, the system requires less memory and thrashing may be resolved
e. may help, if the main memory is increased to a point that all pages of thrashing processes can be resided, the pages
    will no longer be swapped in and out constantly
f. no help, as long as there's no enough memory to reside all these processes pages will go back and forth between
    memory and backing store, faster disk only makes the processes spin faster in thrashing state
g. no help, prepaging will not change the number of pages in a working-set of the program, thrashing still will occur
h. no help, thrashing may be worsen as there's more internal fragmentation in the system, more memory are wasted and
    less can be allocated to processes

9.28
first the page containing the first instruction of the program is brought into physical memory by a page fault, then
the program tries to access a location storing another memory location, the indirect memory access will cause another 2
page faults
the third page fault will replace the page storing the first instruction if the replacement algorithm is FIFO or LRU and
the page is not locked, the process will start to thrash

9.29
this algorithm is almost the same as second chance replacement except that reference bit of each page is reset on its
individual period instead of a global clock
second chance replacement algorithm can be implemented above this mechanism, also the system can reclaim the pages of
a low priority process more frequently, but being a approximation of LRU the hit ratio of this algorithm cannot be
better in general

9.30
a. the initial value of the counter is 0, before a page is loaded into a frame the number of pages associated to a
        frame is 0
    the counter is incremented when a frame is replaced with a new page, a page not loaded to the frame before

the counter is decremented when a page recently being loaded into it is loaded into another frame
the page currently in the frame with smallest counter is replaced

b. ./OS/memory/replacement
CNT: 15 page faults

c. OPT: 11 page faults

9.31
80% * 1 + 18% * 2 + 2% * (1 + 20000) = 401.18 microsecond

9.32
a process has no enough pages to load all of the pages in its working set, causing page faults constantly as a result
the system can record and detect the frequency of page fault of a process, if page fault rate is high the process is
thrashing
more page frames can be assigned to this process or this process has to be suspended

9.33
it usually cannot, whenever a program is working on a set of data in memory it commonly does it in a particular function
or loop, so neither its active code nor data section will change during the working set of data, but there may be rare
cases in which a program works on the same set of data in two different branches of deep and distant function calls,
only move from one branch to another occasionally

9.34
if Δ is too small, the working-set window cannot contain a whole locality, processes will be assigned fewer page frames
than necessary and will start thrashing
if Δ is too big, processes will be assigned too many page frames than necessary, the system will run out of memory
sooner, level of multiprogramming will decrease

9.35
Request 6

```
512
    256
        128
            64
                32
                    16
                        8
```

```
                            6:8
Request 250
    512
        250:256
            128
                64
                    32
                        16
                            8
                            6:8
Request 900
    // not enough memory, rejected
Request 1500
    // not enough memory, rejected
Request 7
    512
        250:256
            128
                64
                    32
                        16
                            7:8
                            6:8
Release 250
    512
        256
            128
                64
                    32
                        16
                            7:8
                            6:8
// other two releases are not allocated at the first place
```

9.36
a working set for each thread, each thread may be accessing different data and executing different function

9.37
the single cache must be shared by all the CPUs, allocation of a kernel object must be protected by mutex and block,
that means process creation, opening files, memory mapping and nearly all syscalls can only be processed one at a time,
diminishes any benefit multiple processors may provide

either each CPU should have their own object cache, or the system has to be asymmetric (i.e. different tasks and cache
are handled on different CPU)

9.38
page size can be decreased to reduce internal fragmentation or increased to reduce the size of page table and the time
and the number of IO operations to load a data set, according to the need of a process
the OS must monitor memory accesses in more details, the translation from logical memory to physical memory using the
same entry in TLB may be illegal or not depending on the current size of the page in that entry

9.39
```
./OS/memory/replacement
1 physical frames
   FIFO: 91 page faults
   LRU: 91 page faults
   OPT: 91 page faults
2 physical frames
   FIFO: 82 page faults
   LRU: 82 page faults
   OPT: 63 page faults
3 physical frames
   FIFO: 66 page faults
   LRU: 72 page faults
   OPT: 49 page faults
4 physical frames
   FIFO: 58 page faults
   LRU: 59 page faults
   OPT: 41 page faults
5 physical frames
   FIFO: 50 page faults
   LRU: 51 page faults
   OPT: 33 page faults
6 physical frames
   FIFO: 45 page faults
   LRU: 46 page faults
   OPT: 27 page faults
7 physical frames
   FIFO: 37 page faults
   LRU: 35 page faults
   OPT: 21 page faults
```

9.40

./OS/memory/collatz-winapi

9.41
./OS/memory/mmu

10.1
even in single-user environment IO may still be issued and waited asynchronously (e.g. in Node.js), a better disk
scheduling algorithm can resolve a batch of IO requests faster so the user process has to wait for less time

10.2
an IO request to a middle cylinder will be handled next under SSTF if it's the closest cylinder inside or outside the
read-write head, but the read-write head cannot move beyond the innermost or outermost cylinder

10.3
the location of sections on a track is managed by the disk controller, not fixed or exposed to the driver software
SSTF: the distance to a sector have to be calculated giving current cylinder & angle of read-write head
SCAN: IO requests on the same cylinder are sorted by their angle
C-SCAN: same to above

10.4
if all IO requests are handled by a single disk or controller, processes waiting for these requests will block longer,
resulting in a lower global throughput

10.5
reading a file by swap space raw IO should be faster than traversing the file system, but swap space occupies extra disk
space and may need to be initialized if this process is the first to use it in the system

10.6
there's always disasters big enough to wipe all backups so no

10.7
a. 1. 512 / (512 / 5 * 2^20 + 0.015) = 33.913 Kbps
   2. 483.019 Kbps
   3. 4.651 Mbps
   4. 4.977 Mbps
b. 1. 0.00647

   2. 0.0943
   3. 0.930
   4. 0.995
c. t / (t / (5 * 2^20) + 0.015) / (5 * 2^20) >= 0.25
   t >= 26214.4 bytes
d. // thanks solutions manual
   K, where K is the block size of the disk
e. cache: 2.237 bytes
   memory: 1.678 bytes
   tape: 40 Megabytes
f. depends on how the tape device is used
   if it's treated as a back up storage of disk then it's a sequential-access device, if it's used as a secondary
   storage (despite of its long access latency) then it's a random-access device

10.8
in a burst of small and random accesses, if the requests are uniformly distributed on all the disks in RAID 1 they can
be handled in parallel, while in RAID 0 they can only be handled sequentially, since the accesses are small the faster
transfer time in RAID 0 will be dwarfed by the long seeking time

10.9
a. SSTF: if there's IO requests to the cylinder around the current position of read-write head constantly, an IO
         request to the inner or outermost cylinder may never be served
   SCAN: if there's an endless stream of IO requests to the same cylinder, the read-write head will stuck at that
         cylinder, requests to other cylinders will never be served
   C-SCAN: same to above
   LOOK: same to above
b. handle one IO request on a cylinder at a time, consequential requests to the same cylinder is handled FCFS
c. a single process generating an endless stream of IO requests can block every other processes which are waiting on IO
   in the same system
d. 1. when the throughput is the only concern e.g. in a batch system
   2. when some IO requests are generated by high priority processes
   3. in a critical situation e.g. during power outage and the system is on limited emergency power, the system should
      write out all processes to swap space and that IO request should be handled with top priority

10.10
SSD has almost uniform seeking time and no read-write head, there's no point to optimize the read-write head movement by
a complicated scheduling algorithm like SSTF, also FCFS ensures fairness

10.11
./OS/storage/disk-scheduling
FCFS: 13011
SSTF: 7586
SCAN: 7492
C-SCAN: 9917
LOOK: 7424
C-LOOK: 9137

10.12
a. d = at^2, t = (d/a)^(1/2), a is constant
b. 1 = x + y
   18 = x + 70.70y
   17 = 69.70y, y = 0.244, x = 0.756
c. ./OS/storage/disk-scheduling
   FCFS: 90.058ms
   SSTF: 65.345ms
   SCAN: 68.873ms
   C-SCAN: 78.277ms
   LOOK: 66.617ms
   C-LOOK: 70.052ms
d. 26.0%

10.13
a. half a round, or 60 * 1 / 7200 / 2 = 0.0167s
b. 16.7 = x + yL^(1/2)
   L = 4269.873

10.14
pros:
   faster random-access speed and transfer speed, no need for complicated scheduling algorithm, more tolerant to
   disasters, uniform seek time, can be put to sleep when idle (the disk must spin whenever power is on)
cons:
   lower expected lifetime, limited writes, more expensive per gigabyte

10.15

ignore transfer time or assume transfer time is proportional to the distance traveled, ignore rotation latency, both
initial position p and IO request r are uniformly distributed
let e be the innermost cylinder

```
   SCAN:   int(p, 0, e, int(r, 0, e, d(p, r)))
        where d(p, r) = r - p if r >= p
                      = (e - p) + (e - r) = 2e - p - r if r < p
        = int(p, 0, e, (2e - p + 2e - 2p)p/2 + (e - p)^2 / 2)
        = int(p, 0, e, 2ep - 3p^2/2 + e^2/2 - ep + p^2/2)
        = int(p, 0, e, ep - p^2 + e^2/2)
        = 2e^3 / 3
   C-SCAN: int(p, 0, e, int(r, 0, e, d(p, r)))
        where d(p, r) = r - p if r >= p
                      = (e - p) + e + r = 2e - p + r if r < p
        = int(p, 0, e, (2e - p + 2e)p/2 + (e - p)^2/2)
        = int(p, 0, e, 2ep - p^2/2 + e^2/2 - ep + p^2/2)
        = int(p, 0, e, ep + e^2/2)
        = e^3
```

C-SCAN have longer expected response time
when requests are sparse C-SCAN may suffer from greater response time variation, about half of the requests must wait
for the read-write head to go back to cylinder 0 and start over again, but when the IO queue is crowded and higher
rotational latency C-SCAN is more fair and has lower variation as requests to the other end have to wait all the other
requests elsewhere to be served

10.16
a. SSTF would be far better than all the other algorithms, other algorithms must go towards a fixed direction so cannot
   focus on a small area
b. SSTF but waits for a small period of time before leaving the hot spot, so the upcoming requests to the small area
   have a higher chance to be handled rapidly

10.17
a. one block, the block of data
b. 9-10 blocks, 7 data blocks and 2-3 parity blocks

10.18
a. RAID 5: read operations on a single block access only one block on a single disk
   RAID 1: same to above
b. RAID 5: contiguous blocks are interleaved to different disks and can be read in parallel
   RAID 1: contiguous blocks are stored in the same disk, can only be read sequentially

10.19
RAID 1: each write operation writes to a single disk
RAID 5: each write operation writes to at least 2 blocks on 2 different disks, global throughput will be lower than
    RAID 1, but write to contiguous blocks can be handled in parallel if there are enough disks

10.20
according to 10.18 and 10.19, file that is frequently read should be stored in RAID 5, while file that is modified by
processes should be stored in RAID 1

10.21
a. once per month, 750000 / 1000 = 750 hours = 31.25 days
b. // thanks solutions manual
   MTBF = 24 * 365 * 1000 = 8760000 hours
   nothing is given about the expected lifetime
c. warranty of a product is decided by financial consideration and legal requirement

10.22
sector sparing:
   less available space, a hard failure of a sector would not reduce the number of available disk space so the disk can
   maintain the labelled spec for a longer period of time
sector slipping:
   less available space, a failed sector will always be mapped to the next available sector, the failure handling
   controller is more complicated, an entire track have to be read and write to different position on hard failure

10.23
when the secondary storage is backed by RAID 5, the OS may want to ensure that a frequently accessed file has its blocks
scattered around all the disks so contiguous read can be performed in parallel
the OS may also require a large file to be stored in the same or close cylinders to minimize the seeking time

10.24
./OS/storage/disk-scheduling
FCFS: 8222052
SSTF: 8110
SCAN: 6884
C-SCAN: 9997
LOOK: 6884

C-LOOK: 9995

11.1
auto reset system may be useful on computers shared by anonymous public users, e.g. in a campus library or internet
cafe, they can use a set of pre-installed softwares while have no access to system files or other users' files
systems that do not reset on user logout is used when the user (partially) owns the computer

11.2
if the system does not provide the API to read / write different type of files according to their content, all user
programs have to parse the files by their own codes or (possibly different) third-party libraries, an approach more
bug-prone
a typed file API on the other hand may be misused or show surprising behavior from time to time, e.g. the difference
between wb and w on some old platform, programmers have to be more careful than dealing a uniform binary file API

11.3
same to above, in practice most files are either text or records of fixed size, glibc provides both blocked IO and
formatted IO, the same set of functionality can be built on a stream byte API, the system programmer has to decide
whether the system should provide these API out of box at the expense of bigger image size

11.4
the real file name can be modified to include the path to that file, for example a file log.txt under directory /etc/
can be named /etc/log.txt, as long as each file has a unique path they will have a unique name in the single-level
directory structure, then an external file explorer can simulate the multilevel directory structure from their file name
if the file name is limited to 7 characters, paths would be to long to be included in file names

11.5
without open() syscall, each read and write syscall must take file path instead of file descriptor as argument, each
call to read() and write() must first traverse the file system to find the entry of the file, either the entire file
system must be loaded into memory on startup or read() and write() syscall will be much slower
open() caches file entry in a table, without close() the OS won't know when to remove these entries from the table and

the table will eventually fill up

11.6
// thanks solutions manual
a. if the structure of the subdirectory is stored in the directory file instead of the file system, by modifying the
   subdirectory the users owning the subdirectory may include files which they do not have access to the subdirectory
   and gain access to these files
b. instead of raw read and write provide another set of syscalls to manipulate directories

11.7
a. create a new group, include 4990 users in that group, set group access permission to rwx
b. create a new group, include the 5000 - 4990 = 10 users, deny access to that file by users in that group
   possible on windows platform

11.8
there will be exactly the same number of access permission entires, but the access permissions of a single user is now
specified at the same place, batch processing a user's access permissions no longer requires traversing the file system,
a user can be removed from the system much easier

11.9
even though the file is deleted, uninformed users will refer to the new file by the links as if the file is still there,
an adversary can exploit this scheme by supply a a malicious replacement and void the security of the system
link should contain a hash of the file which is updated on each access, if the file is modified or replaced since last
access through that link OS can detect a hash mismatch and inform the user

11.10
the text described a two-level scheme:
1. a system-wide open file table that stores the cached entry of the opened file, a counter of how many processes are
   accessing it (incremented on open() and decremented on close(), removed when reduced to 0), its position on the
   disk, etc., information that's independent to individual processes,
2. a process-specific open file table that stores the pointer to the system-wide open file table, the current read and
   write position into the file, the operation mode, etc.

but if processes are executed by different CPUs, the system-wide open file table must be protected by a mutex lock,
this scheme may have lower throughput than maintaining a full-fledged open file table per user

11.11
pros:
  once the lock is acquired the protection is absolute, no other user can gain access to the same file
cons:
  the file system is now more dangerous to use, read() or write() syscalls may block a process indefinitely, a
  misbehaving process has the potential to block a big portion of the system by deadlocking while holding the lock of
  a widely shared file

11.12
sequential:
  video / music player, video game
random:
  database, word processor, text editor, web browser

11.13
pros:
  users no longer have to manually call open(), read() and write() can skip the directory structure by hashing the
  file name into a hash table
cons:
  file in the open file table lives much longer than necessary, fewer processes can run concurrently

11.14
buffer the reads ahead, whenever the process calls read() the system reads much more from the file and put the bytes in
a buffer, subsequent reads and writes is served by the buffer instead of the file on the disk, the buffer is written
back to the file only when a read reads beyond the current buffer size or the file is closed

11.15
a video player that allows users to seek to a particular frame of a video, if bit rate of the video is variable the
position of that frame cannot be calculated and must be searched among the file or the index created by the system

11.16

pros:
  most systems allow such link, forbid it would be counter-intuitive
cons:
  when mount point is shared by different devices, for instance different usb storage may be mapped to the same drive
  letter, the link may be invalid or even accidentally point to a different file of the same path

11.17
stated in depth in the text in section 11.3.6
a middle ground is duplicated file with synchronization, e.g. git or other version control software

11.18
pros:
  errors that only exist on remote file system can be specified and prompted to the user, error messages will be more
  informative, the system can handle remote file system only errors in its own context
cons:
  a different set of API must be provided, or the same API for local file system is reused and behaves totally
  different when the file is located in a remote file system, leading to surprises and painful debug sessions

11.19
each write operation must be immediately reflected to the original file in the remote file system, read and write
operations cannot be buffered by the system (the user still can buffer them at their own risk), read and write
operations are much slower and suffer from network conditions more frequently

12.1
contiguous
a. 202, 100 blocks are read and written to the space occupied by the next block, the new block is then inserted to the
  first free block, then the control block is updated to reflect the new length of the file
b. 2 * (100 - i) + 2, if the block is inserted to ith (0 based) place, 2 * (100 - i) blocks have to be moved
c. 2, 1 block inserted to the end, the control block updated
d. 1, the control block is updated to point to the second block with length - 1
e. min(), either all blocks before it or all blocks after it is moved
f. 1, the control block is updated to reflect the new length
linked
a. 2, the new block contains a pointer to the former first block, the control block is updated to point to the inserted
  block

b. i + 2 for i > 0, the block is inserted as the new ith block (0 based), i blocks are traversed, the i-1th block is
    updated and written back, the new block is inserted
c. 102, 100 blocks are traversed, the last block is updated and written back, the new block is inserted
d. 2, the first block is read, the pointer to the second block is extracted from first block and updated to the control
    block
e. i + 2 for 0 < i < 100 (0 based), the i + 1 blocks are traversed, the i-1th block is updated to point to the i+1th
    block
f. 100, 99 blocks are traversed, the 98th block now points to null
indexed
a. 2, the index block is updated, the new block is written out
b. same to above
c. same to above
d. 1, the index block is updated
e. same to above
f. same to above

12.2
even if the system can track and manage these mounts, a user program either cannot or won't be bothered to track all
these duplicated mounts, operations which are supposed to be performed on each file once may be performed multiple times
on files in that file system
if the system in addition allows cascading mount, the same file system can be mounted to a subdirectory of itself,
possibly creating a loop in the file system

12.3
otherwise a special tag must be contained in each block to indicate whether it is free, and the entire disk must be
scanned on startup to initialize the bitmap, when the disk is spacious this initialization will take several hours

12.4
1. whether the file will be accessed sequentially or randomly, linked file do not support random access well
2. the maximum size of the file, huge file usually cannot be allocated contiguously
3. if the file will be accessed in its entirety frequently, storing it in contiguous space may increase effective
    transfer rate

12.5
same to the cluster scheme described in the text, more internal fragmentation but free from external fragmentation

12.6
read can be buffered ahead so subsequent reads can be served by the buffer instead of causing another IO operation,
write can be buffered so multiple write to a single block can be applied once instead of causing an individual IO
operation each time
cache is more expansive than disk and a system with more cache will suffer more significant inconsistency between memory
and disk after disasters

12.7
pros: the system occupies less memory when the functionality are not used
cons: user will experience noticeable delay first time calling these syscalls

12.8
each file system implements an interface defined by the VFS specification, OS can handle syscalls from user programs by
calling VFS functions from whatever file system mounted at the path transparently

12.9
a. exactly the same to linked file, no external fragmentation, bad at random access
b. the OS must choose between external fragmentation and long links of scattered file extends by ignoring free holes
   under a certain size or not, and it would difficult for the system to decide how much space should be allocated to
   the next extend of a file and may delegate the decision to the user
c. no external fragmentation and the efficiency of random access can be optimized according to user requirements

12.10
contiguous: highly efficient sequential and random access, sequential access can be performed in batch
linked: random access has $\Omega(n)$ complexity
indexed: efficient sequential and random access

12.11
in ordinary linked allocation accessing ith block of a file requires $\Omega(n)$ IO operations, with FAT the links can be
traversed in one or fewer blocks, provides better support to random access

12.12
a. if the file system can obtain a list of all the blocks assigned to it from the OS, it can reconstruct the free-block
   list by deducting the blocks allocated to files from the entire list of blocks
b. thanks http://matthews.sites.truman.edu/files/2017/01/chapter12.pdf
   in worst case each directory on the path contains enormous number of files, forcing any access to go through the
   triple indirect pointer, assume the block of the file entry in the directory can be located immediately by a hash
   table
   1 - 5: read inode or root, read the 3 indirect index blocks, read the file entry block
   6 - 10: same to above for /a
   11 - 15: same to above for /a/b
   16 - 17: read inode of /a/b/c and the direct block containing its content
c. doubly link all the free blocks as a circular list, storing pointers to previous and next blocks in the header, each
   single free block can be used as the new first free block pointer after a memory failure

12.13
small file can be allocated small blocks to minimize internal fragmentation, bigger file can be allocated blocks of full
size to minimize the necessary disk operations for sequential access
alongside the pointers to the next block, the size of the current block must be stored in each free block, or blocks of
different size must be managed in different lists, blocks of smaller size may need to be combined to form a block of
bigger size on demand

12.14
write cache: after a crash data in the write cache yet to be written out is lost, causes inconsistency between control
   block and data
free block bitmap, block index and hash table: by using complicated structure the critical section of an IO operation
   is longer, the system is more vulnerable to crash and data inconsistency
grouping and counting: more difficult to reconstruct the free list after a crash

12.15
a. how is memory address translation part of the file system
b. contiguous: 1, the location of block 4 can be computed from the location of block 10
   linked: 6, the links must be traversed one by one from block 10 to block 4
   indexed: 2, read the index block and then block 4

12.16

8KB / 4B = 2048
(12 + 2048 + 2048^2 + 2048^3) * 8KB = 64.031TB

12.17
a reverse map from block number to files can be built by scanning the entire file system, then blocks can be swapped
to move all free blocks to the end of the disk space
1. doing so requires hours of time, during which normal operation cannot be served or is significantly delayed
2. on certain disk devices recompacting and relocation consumes limited read and write tolerance
3. when programmed carelessly the entire recompacting session is a critical section, crash in between will leave the
    disk in an inconsistent state

12.18
1. each time a file system is mounted or unmounted, the cache entries with a corresponding prefix must be invalidated
    to hide or expose the underlying local directory
2. on certain error condition returned by the remote file system calls, the cache entries must be invalidated, e.g.
    a file is moved or deleted on the server
3. the cache may even be flushed periodically

12.19
an OS recovering from a crash can inspect the log to see what IO operation is yet to be performed or halfway done, then
revert to a safe state and apply these operations anew until the log is emptied and the user programs agree with the
OS about the disk state

12.20
a restore operation only read and copy files from two mediums instead of all mediums from the last full backup, but each
backup occupies more space than the scheme in section 12.7.4

12.21
```
$ echo foo > file1.txt
$ ls -li file1.txt
32932572275255899 -rw-rw-rw- 1 *** *** 4 Dec  9 19:41 file1.txt
$ ln file1.txt file2.txt
$ ls -li file2.txt
32932572275255899 -rw-rw-rw- 2 *** *** 4 Dec  9 19:41 file2.txt
$ cat file1.txt
```

```
foo
$ cat file2.txt
foo
$ echo baz > file2.txt
$ cat file1.txt
baz
$ rm file1.txt
$ ls
bin  file2.txt  file3.txt
$ strace rm file2.txt
// ...
unlinkat(AT_FDCWD, "file2.txt", 0)      = 0
// ...
$ ln -s file3.txt file4.txt
$ ls
bin  file3.txt  file4.txt
$ ls -li file*.txt
39687971716484175 -rw-rw-rw- 1 *** *** 4 Dec  9 19:41 file3.txt
83035118130003454 lrwxrwxrwx 1 *** *** 9 Dec  9 19:58 file4.txt -> file3.txt
$ echo foo > file4.txt
$ cat file3.txt
foo
$ rm file3.txt
$ cat file4.txt
cat: file4.txt: No such file or directory
```
the command is translated to

    cat file3.txt

and the file referred by the link no longer exist

13.1

pros:

  hardware implementation executes faster than software implementation in kernel or user space

  the functionality is better encapsulated, user program or the OS is not even aware of its existence

  the code or circuit of the functionality cannot be modified in any means by the host, immune to virus and malware

cons:

  bugs in the implementation can only be patched by switching the hardware or flashing the EEPROM

  the functionality is completely hidden from the OS, no way to cooperate

  developing the functionality on hardware cost more time and expense, more difficult to debug and iterate

13.2
with a single busy bit:
    the host repeatedly scan it until it's cleared, then set the data registers and set the buzy bit
    the controller repeatedly scan it until it's set, serve the operation request and clear the buzy bit

13.3
interrupt-driven serial port:
    the device behind the serial port rarely generates events, e.g. an sensor that captures environmental data every
    few seconds, a thread should not busily wait for the event but let the OS wake it upon an interrupt
polling terminal concentrator:
    when the OS is on a host dedicated to the management of the terminal concentrator(s) or when the concentrator is so
    busy that generate a series of events the interrupt handler cannot deal with

13.4
the driver can poll for a fixed amount of times then ask the controller to interrupt the OS and go to sleep if no event
occurred in the period
pure polling:
    a coprocessor with high frequency and small data input, a request can be served in nanoseconds, interrupt handler
    cannot operate at that speed or feed the coprocessor with data consistently
pure interrupts:
    a sensor reading environmental data every few seconds, process working on such device will be idle most of the time
    and has no time constraint
hybrid:
    a device which generates events in burst e.g. a microphone, with the hybrid strategy the events can be handled fast
    and the process can be put to sleep when idle

13.5
instead of polling in a busy loop and monopolize the core, multiple IO requests can be issued concurrently, controllers
of different IO devices then can handle these requests and copy data to the supplied addresses at the same time
instead of a few registers grouped into a serial port, these controllers must be able to understand (virtual) memory
addresses and be able to modify it

13.6

faster the CPU, more cycles are wasted on polling if the system-bus and device speed didn't catch up, bigger portion of
CPU time is not doing actual work

13.7
STREAMS module is a interface with a write queue and a read queue, can be read or write in unit of bytes or messages
STREAMS driver is a device driver implemented as a series or STREAMS modules

13.8
among the devices of the same type, the ones with smaller buffer should has higher priority or they are more likely to
lose data due to buffer overflow
an interrupt indicating an error should have higher priority than normal interrupts, an error condition is usually more
urgent than normal IO operations
user should be able to assign priorities to interrupts in a certain range so they can control the behavior of the
interrupt handler while not intervene the operation of the kernel

13.9
pros:
  the device registers can be accessed as normal memory addresses, the user programs don't have to follow special
  protocols and explicit syscalls
cons:
  the mapped addresses may be accessed accidentally by e.g. out-of-bound array accesses, a software error can
  escalate to a hardware defeat through the mapped addresses

13.10
a. polling with a small buffer, the thread handling mouse inputs can be executed every few milliseconds and apply all
  mouse inputs in the buffer at once, the user won't notice the difference
b. interrupt with a buffer in kernel space, the user process should yield CPU to others during the long seek time, the
  tape drive on command will copy its content to the buffer through DMA or registers, then the kernel will wake up the
  user process and copy the content to user memory
c. same to b
d. polling with memory mapped IO, GPU works at high frequency, latency from interrupt handling may not be acceptable

13.11

user programs invoke syscalls with virtual addresses, and in most system even though memory address translation does not
go through OS for performance concern, the page table is still managed by the OS, hence the OS can translate virtual
addresses to physical addresses and request IO operations from the device with physical addresses, a whole category of
illegal access error is eliminated by the translation as memory that doesn't belong to the user process won't be present
in its page table

13.12
1. context switches, both switching out when the process it put on wait list and switching in when the process is
   waken up and allocated CPU time again
2. latency introduced by the interrupt handler, e.g. select the correct handler from a chain of handlers, dispatch the
   interrupt through indirect function call
3. generating an interrupt is more time-consuming than set up the data-out registers
4. depends on the priority of the process may have to wait for arbitrarily long on the ready list
5. the current running process may be in a critical section, the interrupt may be masked and delayed until the current
   process terminates

13.13
blocking IO:
   1. when the process can not proceed unless the IO operation is fully complete, e.g. modifying a file
   2. when the IO operation takes a long time and cannot return halfway
   3. when deadline is near, blocking IO has simpler interface
non-blocking IO:
   1. when multiple jobs are to be balanced among the process itself, by non-blocking IO the process can do other jobs
      when the device is not ready
   2. when the IO device is only available occasionally and the process does not want to wait for the next event
   3. in async environment, blocking IO does not cooperate well with the user space task scheduler, the scheduler has
      no way to know whether the task is blocking on IO or is doing meaningful work

13.14
same to async IO (or is it an alternative description of async IO?), the syscalls return immediately and informs the
process later by an interrupt, meanwhile the process can handle other jobs (explicitly or through a user space task

scheduler), improve job balance among the single thread

13.15
the controller must understand memory translation in the same way as the CPU of the host, i.e. the content and format of
the page table, the controller may also need to manage its own TLB for performance, since virtual memory is platform
specific the controller or the driver must contain platform specific codes for a variety of platforms
the controller instead of the OS can do the memory address translation, more works are offloaded to the controller which
is the purpose of DMA from the beginning

13.16
// only message passing since shared data structure is the default control group
pros:

1. IO managers are decoupled from the kernel, the kernel data structures can be kept as internal implementation
   details and be changed without breaking the IO subsystem
2. no data structures are shared among host and devices, the system as a whole is less vulnerable to state
   inconsistency caused by a crash
3. logging is made much easier, rather than artificial log messages the message itself can be recorded or even
   rewound from the log

cons:

1. higher overhead, the message has to be copied from kernel buffer to controller memory or vice versa
2. the size of kernel and IO managers combined is larger, the same set of functionality has to be implemented on
   both sides
3. when an IO manager requires more information from the kernel after an update, extra messages must be designed
   and passed as it has no access to the data structures of the kernel

13.17
// thanks solutions manual, but the reference code is over simplified
// the hardware clock must support canceling, otherwise if the three channels are all occupied, when a new request
// arrived with notification time earlier than the earliest among the currently running 3 channels, the new request can
// not be handled and must be dropped, which is both arbitrary and unrealistic

13.18
see the difference between TCP and UDP

14.1
capability lists consist of <object, right> pairs associated with each domain, access lists consist of <domain, right>
pairs associated with each object

14.2
on deletion of a normal file, the system would simply remove its entry and add all its blocks to the free list, the
content of the file will only be overwritten when the block is reused, in between any adversary with physical access
to the system (i.e. can scan the disk bypassing the file system) would be able to recover the file
but such a scheme is not perfect, due to the nature of magnetic disks the file may still be recovered with high accuracy
after several overwrites

14.3
subset

14.4
$A(x, y) \subseteq A(z, y)$

14.5
a procedure can go down the stack and modify the stack content inserted by other procedures, e.g. edit the return
address of the calling procedure and invoke another malicious procedure

14.6
ring-protection

14.7
capability list or access list, the right structure now contains a counter, each access to that right decrements the
counter, when the counter is 0 the right is removed from the list

14.8
whatever data structure storing the object stores a counter alongside it, the counter is incremented when a process
gains access right to that object, decremented when the process is terminated, once the counter is reduced to 0 the
object is removed

14.9

IO operations in most system are privileged instructions, if user processes can do their own IO they may be able to
modify kernel data, e.g. the swapped out PCBs on disk, and the kernel can no longer assure its own integrity

14.10
allocate the object in a privileged segment and deny access from the user process to the segment

14.11
see section 14.3.3

14.12
no, by switching the access privileges are granted and revoked, only available to a process in domain A temporarily, by
including the process would have these privileges longer than necessary i.e. violates the least-privilege principle
also the switching right is not transitive, domain A cannot grant the switching right to other domains, by including
the rights may be transferred to any domain other than A and B

14.13
each access right entry contains a time interval, the OS only grants the right to the calling process if the system
time is in the time interval (as a result modifying system time must be privileged), the game is hidden from users and
can only be accessed from a wrapper program that checks user's privilege periodically and terminates the game process
if the user no longer have the right

14.14
// thanks solutions manual
instead of raw bytes the hardware tags each object with its type (a bit pattern) and treat them differently, the same
scheme can protect memory from being manipulated by wrong procedures as when the integer typed data is passed as a float
parameter to a procedure the hardware can detect the type mismatch

14.15
when the state of an object changes the access rights to it can be granted or revoked in one place, each object can have
a default set of access rights as a special list entry, capability of a domain is scattered all over the system thus
hard to manage

14.16
the reverse of 14.15: capability of a domain can be managed (copied, shared, transferred to other domain) with ease,
access rights to an object is scattered over multiple lists, all these lists must be traversed when the state of an
object changed

14.17
access rights of one domain no longer has to be subset or superset of access rights of another domain, rights can be
dynamically granted to or revoked from arbitrary set of domains

14.18
// referring Capability-Based Computer Systems, Henry M. Levy, 1984
// https://homes.cs.washington.edu/~levy/capabook/Chapter6.pdf
to implement the least privilege principle, the process cannot be granted all the rights it needs during its entire
lifetime, instead it's granted the capability to call certain (kernel) procedures which in turn access the system
resources in request. among those procedures some of them can be certified as trustworthy so that they gain independent
rights on their formal parameters, passed by the caller. such procedure is called a "type manager" of the type of its
formal parameter. these rights are granted universally on all objects of that specific type, in contrary to the normal
instance-to-instance capability scheme in Hydra.
Multics implements a similar facility to support cross-domain procedure calls, but capability management in Multics is
not as fine-grained as in Hydra, a process can call all procedures with higher or equal ring number

14.19
a procedure / program can only access resources explicitly given to it, an object should not unnecessarily expose its
internal structure, similar to the encapsulation principle in most OOP languages
as well as least privilege principle it can minimize the possible harm a misbehaving or malicious process can do to the
system

14.20
a. a process can access all resources of higher ring number regardless of whether it need them or not, ring-protection
   scheme by itself cannot enforce need-to-know principle

b. enforced by default, capabilities is granted instance-by-instance, on procedure calls parameters are accompanied by
    a bit vector of capabilities, without the capabilities a procedure has no access to the parameter
c. any procedure above a doPrivileged() annotation has permission to resources protected by checkPermissions(), but a
    Java program cannot directly access the memory hence a reference to an object cannot be forged, need-to-know is
    enforced not by stack inspection but the JAVA language

14.21
a JAVA thread in a less-privileged domain can gain access to privileged resources by inserting the corresponding
doPrivileged() annotation into the stack before accessing the resource, bypassing the protection provided by stack
inspection

14.22
// thanks solutions manual
role in the role-based access control system closely resembles domain in the access matrix, but different to access
matrix roles themselves are not managed by the role-based access control facility but passwords, e.g. RBAC cannot grant
a role the ability to switch to another role bypassing the password system

14.23
a user program in a system adhere to least privilege principle may only cause minimized damage to the environment
on programming error or intentionally

14.24
a user program can still cause damage within its privileges

15.1
in C reference to an array is a pointer with no length information (called "array decaying"), in a safer language like
Rust reference to an array is a slice which has length information built-in, out-of-bound access to the array can be
detected in running time and cause an exception instead of undefined behavior, in Rust references can also be declared
as immutable so its content can only be read but not modified

15.2

impossible until the intruder tries to log in with that password, after then the system can inform the user through e.g.
emails each time the account is logged in so the user will notice immediately when the intruder tried

15.3
to protect the system from rainbow table attack: even if the password is in the rainbow table the hash will be different
so an adversary cannot recover the password without computing each entry of the table with the salt appended
denote the password by p, salt by s, let H be a secure hash function, H(p || s) is computed and stored along with the
salt, on user login the system takes the password input pi, compute H(pi || s) and compare it with the stored hash value

15.4
store hashed password instead of plaintext

15.5
// referring Watchdogs - Extending the UNIX File System, Bershad and Pinkerton, 1988
// https://www.usenix.org/legacy/publications/compsystems/1988/spr_bershad.pdf
pros:

1. the watchdog is a separate user-level program, compared to a kernel module it won't crash the system on
   exception, requires less privilege, can be deployed and configured more conveniently
2. instead of an universal security scheme in the kernel, users can define their own watchdogs hence deploy their
   safety measure file-by-file

cons:

1. a watchdog, once linked to a file by a syscall, re-implements other file system syscalls like open() and read(),
   such re-implementation is totally transparent to the processes accessing the linked file, once the system is
   breached, the intruder can e.g. redirect write() syscalls to remote host without being noticed by the user
2. correctness of a watchdog is up to the user who implemented it while most users in practice cannot configure
   a security system accurately, this scheme is more or less asking for trouble

15.6
1. according to wikipedia (https://en.wikipedia.org/wiki/COPS_(software)), COPS is mainly a signature-based detection
   engine, means it cannot prevent new 0-day vulnerability and should not be trusted blindly, other security measures

must be deployed in parallel
2. COPS is a user-level program, any intruder with sufficient privilege can modify its configuration thus circumvent
its detection completely, COPS should be launched from read-only media

15.7
enable as few as possible components of the system on installation, a vulnerability in a rarely used component will not
affect every system in the network
users must be provided more privileges to enable these components, which may lead to further vulnerability and a bigger
attack surface to social networking

15.8
his behavior is clearly violating 18 U.S. Code § 1030, the interesting thing is he hadn't be shut out from the industry
for his conduct and later became a tenured professor in MIT

15.9
1. unauthorized access to accounts should be prevented, physical, human and OS
2. personal information associated with accounts should not be leaked to even employees, physical and OS
3. the system should not be vulnerable to temporary power surge and outage, physical and OS (backups)
4. backup should be stored in safe place, physical and human
5. unnoticed tempering from an employee should not be possible, mainly OS and protocol design
6. the system should be recoverable from mistake of inputs, physical and OS

15.10
1. the data is secured in a broader sense: not only within the system in which the data is protected by access control,
it also cannot be recovered from raw physical access to the disk
2. encryption when implemented right is more bug tolerate: a wrong password cannot decrypt the data, while a system may
grant access right to unauthorized user due to a bug

15.11
1. instant message programs, man-in-the-middle can block, insert or modify the messages: end to end encryption,
authorization and sequential number
2. remote login, if passwords are transferred in plaintext will grant access rights to man-in-the-middle: encryption

15.12
unless the two communicating entities are located nearby (i.e. can be configured by the same administrator) or can
arrange physical or any other secure but slow contact, asymmetric key exchange is always preferred, though it's
impossible when both sides do not have a well established and verifiable identity

15.13
ke is publicly available hence cannot prove the identity of the sender
the encrypted message can only be recovered by whoever possess the private key, secure channel can be established
without a shared secret

15.14
a. if the public key cannot be derived from the secret key, keep the public key secret, publish the secret key, encrypt
    the message with the public key, others can verify the author by decrypting the message
b. normal usage
c. the above two combined with two pairs of public & secret keys

15.15
true alarms: 10 * 20 = 200
false alarms: 10000000 * 0.0005 = 5000
200 / (5000 + 200) = 3.84% alarms corresponds to real intrusions

16.1
1. type 0 hypervisor, bundled with the hardware in the form of firmware, guest OSs are usually assigned dedicated
    hardware resources (a hardware partition), supports only a few number of guest OSs or type 1 hypervisors
2. type 1 hypervisor, loaded by a boot loader in place of a normal OS, runs in kernel mode, shares hardware resources
    among guests as if the guests have dedicated ownership, manage hardware resources just like ordinary OS, achieve
    efficiency with support from the hardware
3. type 2 hypervisor, launched as a user process, runs in user mode (occasionally injects kernel mode code into the OS
    by fake device drivers), run guest OSs as normal user processes, cannot benefit from hardware support

16.2
1. paravirtualization, normal hypervisors gives guests the illusion that they are running on their own system, but

paravirtualization provides a non-standard hardware interface and requires cooperation from the guest OSs in hope
to share the hardware resources more effectively
2. programming environment virtualization, instead of virtualizing hardware and run guest OSs it provides a common
interface to user programs so they can invoke the same set of functionalities on different OSs and hardwares
3. emulation, instead of providing a virtualized set of the same hardware resources as the underlying host, an emulator
simulates a different system / hardware to softwares built for that system / hardware
4. application containment, provides virtualization of operating system instead of hardware resources, manages user
land programs instead of guest OSs, give programs the impression that they are the only program running in the
system

16.3
1. protects the host from the guest operating systems by another layer of indirection between them
2. guest operating systems can be suspended, cloned and restored from backups with ease
3. cloud computing server can be sold in a standardized unit regardless of the capability of the hardware
4. enables live migration

16.4
on some CPU there's no clean separation between privileged and non-privileged instructions, e.g. an instruction may have
different execution result in kernel mode and user mode, as guest OSs runs in user mode, trap-and-emulate is only
possible if all privileged instructions results in a trap to the hypervisor through the host
the VMM can perform a binary translation: when the guest executes a privileged instruction in virtual kernel mode,
VMM examines each instruction beforehand and translates them to equivalent instructions on the real machine

16.5
1. new execution mode, appears to be full-privileged to a guest OS but actually passes control to the VMM on privileged
instructions
2. hardware support for virtual page tables, the MMU can go through both page table of the guest OS and the page table
of the VMM, mapping virtual addresses in guest OS to physical addresses without interfere from the VMM

16.6
on virtualized system a guest OS is managed by the VMM, which can easily suspend the guest OS and copy its states to
another host, but on native system the OS must suspend itself and then copy itself to another machine, its state is not
managed by a separate entity thus may be hard to collect, e.g. trying to save the kernel CPU registers may modify the
content of them

17.1
by forwarding the broadcast packets to all ports except the source, the number of packets in all but tree shaped
networks increases exponentially, such a broadcast storm will quickly congest all the links
in a small tree shaped network a broadcast packets can reach all the hosts if forwarded by the gateways

17.2
pros:
  cache hit saves an RTT to the DNS
  functional even when the DNS is temporarily offline
cons:
  causes inconsistency when the entry on DNS is updated

17.3
pros:
  once the circuit is established, transition is much more stable than packet switching and the link is free from
  congest if transmission rate never exceeds the capacity, packets arrives in order with constant delay
cons:
  extra initial cost to establish the circuit, allocated link capacity cannot be recycled when the circuit is idle,
  each link on the circuit is a single point of failure
viable usage: VoIP and local live video transmission

17.4
1. a transparent distributed system should provide an interface which does not distinguish between local and remote
   resources, act like a conventional centralized system
2. a transparent distributed system should provide user mobility by bringing over users' environment to wherever
   they log in, allow them to access from more than one terminal machine

17.5

a. the same method to move a process from one processor to another but over a network and the opened file should be
    available in both hosts or be moved along the PCB
b. if the user process runs in a virtualized environment (e.g. JVM) the process can be migrated as usual as long as
    the two systems have compatible file system, otherwise the process migration cannot be transparent and the user must
    explicitly program how to transfer the state and resume processing on another host

17.6
a. 1. link failure, the network link between client and server failed
    2. site failure, the server is offline and cannot handle requests from the client
    3. packet loss, a message is not transmitted from one site to the other
b. both 2 and 3 are possible in centralized system, a server process may be terminated by a fatal error, loop back
    packets may be lost due to temporary failure in network adaptor

17.7
some active sensors broadcast their measures periodically no matter there is a host accepting their broadcasting or not

17.8
a. only if there's multiple other sites monitoring the condition of B, A can contact them
b. only if there's another link between A and B and they can confirm by switching to the new link
c. indistinguishable from B goes down
recovery from failure may cause further problems, a better solution is to switch to another server on any of the failure
listed above if available

17.9
computation migration is usually visible to user programs in term of RPC calls to execute a computation on another site
and receive the result later, while process migration can be transparent to users such that the user processes can be
move to a different site and been executed there

17.10
OSI model is quite an over specification: the session and presentation layer are not always necessary for a protocol,
in practice handshaking and data representation are up to the application, while the link and physical layer are
implemented in hardware so is not of the operating system's concern
by eliminating the presentation layer applications have to define and parse their messages explicitly and may cause

error e.g. encode a number as little endian and parse it as big endian

17.11
without congestion control doubling the speed of the system doubles the number of packets pushed to the link and router,
possibly overflows their buffer and cause packet loss, the lost packets must be retransmitted and make the congestion
more severe, a less fraction of packets transferred would be original
proper congestion control is necessary when the link may be at capacity

17.12
the same set of functionalities can be implemented with just enough hardware or outright be implemented in hardware,
greatly reduces the cost of the device and boost service stability at the same time as there's no other processes
running on the device
the downside is the device may be harder to upgrade and configure at the time, recently most routers can be configured
from a web interface

17.13
pros:
  when the result of name resolution is not static, name server can be updated once and for all, while static tables
  on each client must be reconfigured
cons:
  name server without backups is a single point of failure, also instead of a table in secondary storage an entire
  protocol must be designed and deployed on both client and server, name resolution will be several order of
  magnitudes slower than table access
each client can maintain their own cache of name resolution, skip the RTT on cache hit

17.14
so the protocol has a clear start point and a change can be contained in a small scope
if there's no top level name servers, name resolution requests would have to be propagated all over the network just
like routing information, take minutes instead of fraction of a second
if there are only top level name servers, any change to any domain name must be committed to the top name servers, the
servers would be flooded by modification requests and cannot reply to requests when the resolution table is
being updated

17.15
pros:
  absolute and transparent reliability
cons:
  when the upper layer does not need reliability, the additional overhead of acknowledgement, error checking &
  retransmission is of no purpose
  physical layer protocols put restrictions on mediums, more strict protocol limits the mediums the entire network
  stack can traverse

17.16
with dynamic routing packets may arrive out of order, if the order is of importance the packets arrived from network
must be buffered and reordered, by the application or by a transport layer protocol e.g. TCP
when latency must be minimized (e.g. telephone, remote medical operation) virtual routing is more appropriate than
dynamic routing

17.17
// skipped

17.18
UDP is not reliable, packets may be lost or reordered on arrival, while a single byte of error in HTTP documents may
render the entire page malformed
one possible solution is connection multiplexing in HTTP/2, once a TCP connection is established multiple data stream
can be carried by it independently, the connection is closed only when all the streams are terminated

17.19
pros:
  the API is compatible with a centralized system, user programs written for a centralized system can be compiled with
  or dynamically linked to the system library
cons:
  a distributed system is inherently different to a centralized one, with shortcomings and benefits the users are
  unaware of, for instance the user may assume a file in secondary storage can be accessed in milliseconds but in fact
  it's located in a remote host and takes seconds to open

17.20

1. remote access, the file system can be accessed from multiple terminals, the home environment of a user can be bought
    to a remote site
2. file name in DFS may be independent to its location, an access to a file can be served by one of many servers
    holding copies of that file, higher availability and fault tolerance
3. capacity of the file system and be extended transparently by connecting another server machine to the network

17.21
NFS is more suitable, since database accesses are typically random and OpenAFS by default uses a bigger cache block,
remote database in OpenAFS will experience more cache miss on the same amount of cache memory, also cache are only
synchronized to server on file close, database in OpenAFS have to deal with inconsistency and merge conflict more
frequently

17.22
NFS is partially location transparent: a remote directory can be mounted to arbitrary local mount point and appear as a
local directory, but the internal structure of the remote directory is exposed hence this scheme is not location
independent at all
OpenAFS by associating each file with a location-independent file identifier provides both location transparency and
location independency, file names are mapped to identifiers then to physical locations

17.23
// cannot think of a situation in which user requires only location transparency but not location independence

17.24
// thanks solutions manual
failure recovery and RPC as a whole is no longer necessary, the same interface of local procedure call can be reused
across the network
virtual circuit should be preferred as there's no link failure and packets may still arrive out of order in dynamic
switching, once the virtual circuit is established it's solid
if the links are also fast client-side cache can be omitted

17.25
// referring http://pages.cs.wisc.edu/~remzi/OSTEP/dist-afs.pdf

all AFS clients launches a daemon called "Venus process", cache of a file is assumed to be valid until the server
contacts the Venus process of a client about an update by another client or the server itself, when reopening a cached
file the client sends the server a validity check request, if the request timed out because of server failure the cache
is discarded and the client will request a new copy of the file from the server

17.26
local and remote cache serve different purposes, local cache ensures that most access to a remote file can be handled
locally, remote (in memory) cache brings data from secondary storage to main memory so it can be read or written many
order of magnitudes faster by the server

17.27
1. whole-file caching: the entire file is transferred on file open and only synchronized on file close, most file
   operations performed locally to reduce the server load
2. file status callback: the server notifies the clients on file modification, no heart beat packets are transferred
   from clients to server

17.28
// the relative text must be edited out, the only mention to the Apollo Domain is using a timestamp as one part of the
// file name

17.29
whole-file caching vs periodical write back
stateful (callback list) vs stateless

18.1
cons:
1. a subset of these modules must be loaded at startup which means longer launch time than a monolithic kernel
2. a malicious program may be able to escalate its privilege by injecting kernel modules to the system

when the OS works on a fixed hardware it can be compiled into a single binary file, otherwise it should always be split
into modules or the OS cannot utilize new hardware released after its compilation

18.2
// thanks solutions manual

1. threads can be managed completely in user mode (N:1 model), from the viewpoint of the kernel it's a single process
   while the CPU allocation and context switch among threads of a process is scheduled by a user mode library
2. threads can be managed directly by the kernel (1:1 model), e.g. in Linux each thread is a task as well as a
   single-threaded process
3. a hybrid approach (M:N model), software threads are mapped to (usually fewer number of) hardware threads, the
   mapping is controlled by a user mode library while the CPU allocation and context switch is handled by the kernel

in Linux threads are managed by the kernel (model 2), threads of a process can be executed concurrently on multiple
CPUs, which is impossible for software threads since they are transparent to the kernel, by contrast software threads
takes less time to initialize and switch, also when done correctly software threads can still benefit from asynchronous
and non-blocking IO

18.3
pros:

1. calling kernel procedures or accessing kernel data structures won't cause page fault, these operations will have
   a lower and more consistent delay
2. the OS will not be preempted by page fault while updating the kernel data structure, less synchronization

cons:

1. the kernel consumes more than necessary memory as even a kernel procedure that's never used will never be paged
   out, user processes will start thrashing earlier
2. the kernel has to manually release pages once occupied by a variable size data structure

18.4
pros:

1. only one copy of shared library has to be stored in secondary storage and main memory
2. dynamically linked library can be updated without recompiling the main program
3. the same source file can be compiled on different platforms given a dynamically linked libraries as a compatible
   interface

cons:

1. linking errors are the one of most prevalent and hardest to fix errors during software development and
   deployment (e.g. user may update their libraries to a new version that's incompatible with the program), a

single statically linked binary is guaranteed to work
2. for security concern: the shared library may not be what the program is expecting

18.5
networking sockets:
1. the network stack is standardized by RFCs and available on most platforms, the communicating programs don't have
to contain code for each different platform just for communication
2. network sockets are full-duplex, both sides can send and receive data at the same time free from conflict and
data race

shared memory:
1. it is much faster to write to and read from the shared memory pages instead fo going up and down on the network
stack
2. communication by sockets must copy the data from buffer to buffer, while shared memory contains a single copy of
the data

18.6
// thanks solutions manual
1. modern disk hardware contains on-board controllers which optimizes and reorders disk read and write, no need for the
kernel to do these jobs
2. the geometry of modern magnetic disks are complex and varies from manufacturer to manufacturer, almost impossible to
optimize read and write for all of them

18.7
pros:
1. ensured type safety, higher level of abstraction, less human mistakes
2. automatically prepare stack for procedure calls, align structures and allocate local variables
3. much more readable and easier to write

cons:
1. high level languages are cross platform, system specific operations e.g. saving registers to PCB can only be
implemented in assembly
2. the compiler occasionally will not optimize the code to the most efficient assembly

18.8
when the environment of the new process must be prepared with heavy computation and exclusive read-write access to the

data in the memory of the parent process, by calling fork() then exec() the environment can be prepared concurrent to
the parent process
when the environment of the new process is immediately available vfork() is more appropriate, otherwise modifying about
anything between exec() and vfork() is UB

18.9
file transfer: TCP, each single bit may be significant to a file, the socket must be reliable
periodical test: UDP or raw ICMP packets, TCP makes too much assumptions and an outage is only visible to application
layer after TCP timeout (1-5 seconds)

18.10
// again an exercise seemingly referring removed section
layered design & generic interface

18.11
pros:
  encapsulation, a kernel module cannot access arbitrary code in the kernel
cons:
  the additional effort to maintain the symbol table exported by the kernel and kernel modules

18.12
1. prevent modules from having unprotected concurrent accesses to a hardware
2. to make the job easier for module loading utilities e.g. modprobe & autoprobe

18.13
clone() syscall accepts flags which specifies what are shared between the parent and child process / thread, if no flag
is set the child shares nothing with the parent thus is equivalent to a child process, if file system, memory, open file
table and signal handlers are all shared then the child is equivalent to a thread

18.14
kernel-level threads, Linux do not distinguish between process and thread in regard to scheduling, all Linux threads are
visible to the kernel and scheduled by the kernel on waiting or ready list

18.15
the objects not shared between child and parent processes - open file table, memory pages and signal handler table -
must be copied on creation

while there's no difference between the scheduling of processes and threads as Linux does not distinguish between them,
switching to another process costs much more CPU time than to another thread as the objects must all be loaded from
backup storage

18.16
traditionally UNIX systems allocate a fixed time slice to each process and allow them to run for the time slice no
matter how much processes are there in the system, therefore the more processes running in a system the longer a process
must wait before its turn to come, by contrast CFS, the new scheduling algorithm introduced in Linux 2.6, allocates 1/N
of CPU time to a process when there are N running processes and each process is allowed to run once every target latency
, reduces the response time of an interactive process when the system is crowded

18.17
target latency and granularity
small target latency & small granularity:
  lower latency, more time wasted on context switching
large target latency & large granularity:
  higher latency, processes run for longer before being preempted, more CPU time on actual job

18.18
Linux kernel offers no guarantees about how quickly a process will be scheduled once the process becomes runnable, to
enforce that each process must specify a deadline and the scheduling algorithm must order the runnable processes by
in respect to their deadline and report an error if it's impossible to meet all the deadlines, thus the systems may not
be able to handle the same amount of processes as before

18.19
when a user process requests access to page not backed by a file, e.g. heap allocation

18.20
when a private page is shared between parent and child processes

18.21
pros:
1. code in shared memory is less critical than kernel code, error in shared library will neither crash the system

nor escalate to fatal security breach
2. by loading shared library on demand memory consumption of kernel is reduced
3. shared library can be extended easily and can be more stable

cons:
1. an additional level of indirection, less efficient compared to a single kernel binary

18.22
pros:
1. after a crash the system can replay or undo operations and restore consistency
2. the OS can commit IO operations much faster, operations in a transaction can be performed asynchronously if the
driver and controller of the hardware doesn't support it already

cons:
1. another level of indirection

metadata operations benefit more from the journaling system as multiple operations can be performed in batch

18.23
the implementation of file systems are hidden behind the virtual file system interface, the kernel does not have to
change to support new file systems

18.24
1. setuid in Linux allows a process to drop and reacquire its effective UID repeatedly, effective UID of a process can
be set to real UID and restored to previous value while in UNIX the only operation possible is switching real and
effective UIDs
2. fsuid and fsgid, a independent UID & GID pair only used on file accesses
3. right passing between programs

18.25
// again an exercise seemingly referring removed section
// thanks solutions manual
1. vast amount of people examine the source code of Linux kernel so vulnerabilities may be found earlier than closed
source projects (not always true, see OpenSSL/heartbleed)
2. people with malicious intent will find the vulnerabilities earlier too
3. patches to vulnerabilities can be deployed faster, everyone can download the new source code and compile instead of
waiting for the official release of a new version

19.1
32/64-bit preemptive multitasking client operating system

1. supporting both Win32 and POSIX API by client-server architecure
2. shipped with a GUI

19.2
1. security: access rights are controlled by ACLs, granted to users or user groups, in additional to a capability
   mechanism called integrity level similar to rings in Multics
2. reliability: code is tested with fuzzing (randomly generated input to programs), memory diagnostic at boot time,
   collect user statistics for timely patch
3. Win32 and POSIX dual API compatibility
4. performance
5. extensibility
6. portability
7. i18n
8. energy efficiency
9. dynamic device support

19.3
1. the hardware boot loader is loaded from on board ROM, runs POST diagnostics, sets devices to clean state, load
   windows bootmgr from the system disk
2. if the system was hibernating winresume restores the state of the system, otherwise bootmgr loads winload
3. winload loads the hardware-abstraction layer, the kernel and device drivers, execution transferred to the kernel
4. kernel initializes itself, starts system process and SMSS initial user process
5. SMSS loads more drivers, initializes paging files, starts session 0 for system-wide background services, execution
   transfers to WINLOGIN
6. WINLOGIN logs on a user, starts EXPLORER the GUI process, LSASS the security subsystem, SERVICES the service control
   manager and CSRSS the Win32 environmental subsystem

19.4
1. the hardware abstraction layer (HAL) that hides the difference between chipsets of the same architecure
2. the kernel which handles thread scheduling, low-level processor synchronization, interrupt and exception handling,
   and switching between user and kernel mode
3. the executive that contains object manager, virtual memory manager, cache manager, security reference monitor,
   plug-and-play and power managers, registry and booting

19.5
managing kernel entities, including processes, threads, semaphores, mutexes, files, sections, ports and other interal
IO objects, generating handles when an object is created or opened, checking whether a process has the rights to access
an object, enforce quotas e.g. the maximum amount of memory allocated to a process, maintaining the internal name space
of objects

19.6
1. creating processes and threads with priorities and processor affinities
2. deleting threads
3. support debuggers by providing APIs to suspend and resume threads or to create threads that begin in suspended mode
4. attach a thread to another process
5. thread impersonation

19.7
a mechanism of full-duplex message passing between a server and a client process on the local system, used by local RPCs
and native Windows services, ALPC supports three message passing techniques: copying data, passing data address and
length, and shared memory

19.8
managing file systems, device drivers, network drivers, buffers of IO devices, controlling the cache manager, working
with the VM manager to provide memory mapped file IO, notify processes for IO events by APCs

19.9
Windows supports both P2P and client-server networking, windows implements transport protocols as drivers in the driver
stack, SMB is a protocol for remote IO requests, PPTP is a protocol for communicating between remote-access server
modules and supports encryption

19.10
as a hierarchy of directories, file entries in each directory is stored in a B+ tree

19.11
in NTFS all file system data structure updates are performed inside transactions, a transaction records redo and undo

information before altering the data structures, after a crash transactions are redone and undone so as to ensure the
metadata is restored to a earlier consistent state, but user files are not protected by this scheme

19.12
same to 19.13

19.13
1. reserve and commit virtual memory by VirtualAlloc()
2. map files to virtual memory by CreateFileMapping()
3. allocate heap memory by HeapCreate(), HeapAlloc() and HeapRealloc()
4. allocate thread local global variables by TlsAlloc()

19.14
when a thread's time quantum runs out the scheduler preempts it by a DPC which is a kind of software interrupt, also to
block the handling of interrupts beside those generated by urgent IO devices

19.15
handle is a reference to an object / kernel entity in kernel space, stored in the handle table of the process, a handle
can be acquired by creating or opening an object, by receiving a duplicate from a communicating process, or by
inheriting from the parent process

19.16
on IA-32: memories are divided to pages of size 4KB or 2MB, the upper 2GB are mapped to system code and data structures
and are inaccessible by user code, including self-map (page table of the process mapped to virtual address), hyperspace
(process specific working set information) and session space (drivers shared among processes in a session), the lower
2GB user memory can be allocated by first reserve virtual pages then commit pages to physical memory
to improve performance, page table in virtual memory of each process always occupies the same virtual addresses, write
out pages by LRU policy, and faults in adjacent pages when a page is faulted

19.17
addresses right after the address allocated by malloc() can be marked as no-access, so buffer overflow will result in
an exception instead of being unnoticed

19.18

1. by copying the data from sender to receiver memory, using the port buffer as an intermediate storage
2. by setting up a shared memory section, both sender and receiver can read and write the section
3. by exposing part of the virtual memory of the sender to the receiver

the communicating channel should choose among the methods by data size of messages

19.19
the cache manager
cache in Windows is divided into blocks of 256KB, each can hold a view of a file, each IO request to a file marked as
cacheable and invalid (not in cache) is passed from IO manager to cache manager, which in turn allocates a cache block,
maps a view of the file into that block, then copies the block into user's buffer
the cache manager predicts the access and prefetches blocks into the cache, and by default dirty blocks are written back
every 4-5 seconds

19.20
file entries of a directory is stored in a B+ tree instead of a table

19.21
a process is a program in execution, the unit of work in a system, a kernel object in Windows with its own virtual
memory space and handle table, a process is created by the CreateProcess() syscall and may contain one or more threads
with their own stacks and registers

19.22
1. fiber is a user-mode facility, kernel is unaware of fibers but threads are scheduled by the kernel
2. fibers are cooperatively scheduled, only explicitly yield the thread to other fibers instead of been preempted
3. fibers shares the same thread-environmental block

19.23
1. fiber is a unit of execution defined in the scope of a single thread, while UMS schedules different user threads to
   the same kernel thread
2. fiber is exposed to the user programs while UMS is hidden from users, may only be utilized by system libraries
3. fiber is unsafe to use because they share the same TEB, UMS does not have such problem

19.24

a UT that will normally block on IO requests waiting for its KT can then explicitly yield the execution to another UT or
be scheduled by a kernel mode primary thread, basically why UMS is possible

19.25
// again an exercise seemingly referring removed section

19.26
Windows uses a multilevel page table for each process, beside the top level not all entries in all levels are valid,
a page-directory entry (PDE) will only point to a valid in-memory PTE or PDE table only when the corresponding virtual
addresses in that range are reserved

19.27
with self-map page table itself can be accessed from virtual addresses, can be paged in and out as any other pages in
the memory space, is backed by paging files or a regular file like normal pages

19.28
it won't work, the entire content of memory is stored on disk and brought back when the system leaves hibernation, if
the amount of physical memory shrank or the number of register changed the same set of pages and registers cannot be
restored

19.29
the suspend counter can be used to simulate a semaphore: a thread can create multiple sub threads and be resumed only
when all sub threads have done their job and resumed the parent thread

20.1
as system libraries e.g. FORTRAN compiler and assembler is stored on magnetic tapes and must be manually loaded and
unloaded on demand, jobs that require similar libraries are executed in batch by the operator

20.2
magnetic tapes are inserted between input card reader and CPU, as well as between CPU and card printer as buffers, which
are later replaced by magnetic disks that can be written to by multiple card readers and read from by multiple output
devices to match the speed difference between CPU and IO devices

20.3

the page replacement algorithm on Atlas assumes that programs access memories in loops, it uses the two most recent
reference to a page to predict the next access to that page, if a page is not accessed by the predicted time it's
written out to backing storage, age of a page is not tracked or decremented

20.4
1, 2 then 4

20.5
Mach can replace BSD with any other operating system interface or execute multiple operating system interfaces at the
same time

20.6
features of operating systems on huge computers will eventually be available in operating systems on smaller computers
or even mobile devices