CS 350
Personal Notes

Marcus Chan

Prof: Kevin Lanctot

Topic 1:
Introduction to Operating Systems

An OS is "part cop, part facilitator".

An OS is a system that

① manages resources;
eg processor, memory, I/O

② creates execution environments;
eg interfaces → resources

③ loads programs; &

④ provides common services & utilities.

THREE VIEWS OF AN OS

3 views:

① "Application view" — what services does it provide?

② "System view" — what problems does it solve?

③ "Implementation view" — how is it built?

APPLICATION VIEW

The OS provides an "execution environment" for running programs.

More specifically:

① Provides a program with resources that it needs to run;

② Provides interfaces through which a program can use networks, storage, I/O devices and other system hardware components; &

③ Isolates running programs from one another and prevents undesirable interactions among them.

SYSTEM VIEW

The problems an OS solves:

① It manages the hardware resources of a computer system;

② Allocates resources among running programs; &

③ Controls the access to or sharing of resources among programs.

IMPLEMENTATION VIEW

The OS is a concurrent, real-time program.

CONCURRENT

"Concurrent" means multiple programs or sequences of instructions running or appearing to run at the same time.

Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.

REAL-TIME [PROGRAM]

A "real-time" program is a program that must respond to an event in a specific amount of time.

eg when you are watching a video, you do not want it to freeze up or skip.

KERNEL

The "kernel" of an OS is the part of the OS that responds to system calls, interrupts & exceptions.

Intuitively, it is the part of the OS that is always running from start-up to boot-down.

OPERATING SYSTEM

The OS on a whole includes the kernel, and may include other related programs that provide services for apps.

These include

① Utility programs;

② Command interpreters; &

③ Programming libraries.

MONOLITHIC KERNEL

A "monolithic kernel" has "everything" as part of the kernel, including device drivers, file system, virtual memory etc.

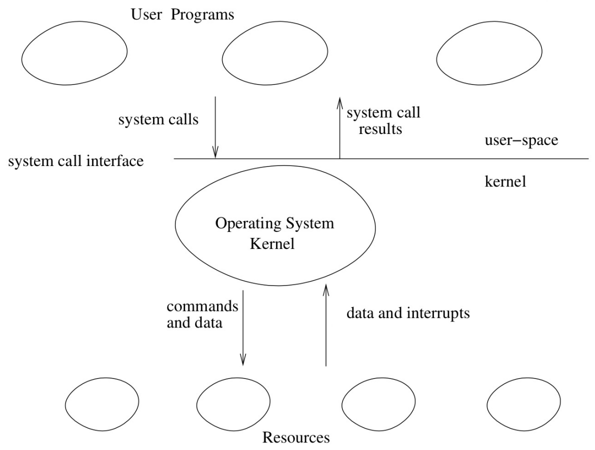
MICROKERNEL

A "microkernel" includes only absolutely necessary components; all other elements are user programs.

REAL-TIME OS

A "real-time OS" is an OS with a stringent event response time, guarantees, & pre-emptive scheduling.

SCHEMATIC VIEW OF AN OS



USER PROGRAMS

"User programs" are programs that the user interacts with directly.

eg desktop environments, web browsers, games, editors, compilers, etc

USER SPACE

The "user space" refers to all programs that run outside the kernel.

These programs do not interact with the hardware directly.

SYSTEM CALL

A "system call" is how a user process interacts with the OS.

KERNEL SPACE

The "kernel space" is the region of memory where the kernel runs.

OS ABSTRACTIONS

💡$_1$ The execution environment provided by the OS includes a variety of abstract entities that can be manipulated by a running program.

💡$_2$ Examples:

① Files & file systems

↳ secondary storage (HDD/SSD)

② Address spaces

↳ primary memory (RAM)

③ Processes & threads

↳ program execution (processor cores)

④ Sockets & pipes

↳ network & other message channels

Topic 2: Threads and Concurrency

THREADS

1 A "thread" is a sequence of instructions.

2 A "normal" sequential program consists of a single thread of execution.

3 A program can have multiple threads or a single thread of execution.
↳ similar to NFA (set of current states) vs DFA (one current state)

4 In particular, a single program can have
① different threads responsible for different roles;
② multiple threads responsible for the same roles.
eg web-server might have 1 thread for each person using the site

CONCURRENCY

1 "Concurrency" occurs when multiple programs / sequences of instructions make progress;
ie they run / appear to run at the same time.

2 Threads provide a way to express concurrency in a program.

PARALLELISM

"Parallelism" is when multiple programs or sequences of instructions can run at the same time.
ie there are multiple processors / cores.

WHY THREADS?

1 Advantages:
① Efficient use of resources
- one thread can run whilst the other waits
② Parallelism
- diff threads can run on diff cores
③ Responsiveness
- one thread can be responsible for a long task
④ Priority
⑤ Modular code

2 A thread blocks for a period of time or until some condition has been met; concurrency lets the processor execute a different thread during this time.

thread_fork

1 A thread can create new threads using `thread_fork`.

2 New threads start execution in a function specified as a parameter to `thread_fork`.

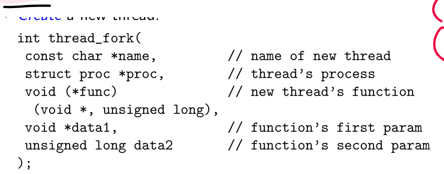
3 The original thread & new thread now proceed concurrently as two simultaneous sequential threads of execution.

4 Note:
① Each thread has its own private stack;
② All threads share access to the program's global variables and heap; &
③ In the OS, a thread is represented by a structure/object.

Examples: kern/test/threadtest.c;
kern/test/traffic.c.

OS/161's THREAD INTERFACE

1 Create a new thread;



```
int thread_fork(
  const char *name,            // name of new thread
  struct proc *proc,           // thread's process
  void (*func)                 // new thread's function
    (void *, unsigned long),
  void *data1,                 // function's first param
  unsigned long data2          // function's second param
);
```

*see kern/include/thread.h for details.

2 Terminate the calling thread;

```
void thread_exit(void);
```

3 Voluntarily yield execution;

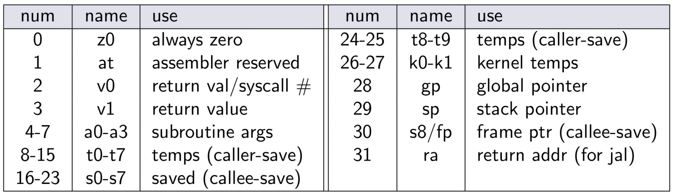
```
void thread_yield(void);
```

4 Note: we cannot control the order that the threads run in without using additional tools.
eg synchronization primitives.

SEQUENTIAL PROGRAM EXECUTION

A single threaded program uses the "fetch-execute" cycle.

MIPS REGISTERS FOR OS/161



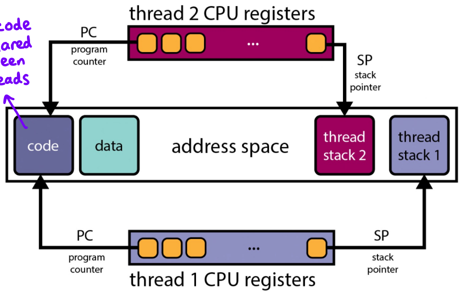
num	name	use	num	name	use
0	z0	always zero	24-25	t8-t9	temps (caller-save)
1	at	assembler reserved	26-27	k0-k1	kernel temps
2	v0	return val/syscall #	28	gp	global pointer
3	v1	return value	29	sp	stack pointer
4-7	a0-a3	subroutine args	30	s8/fp	frame ptr (callee-save)
8-15	t0-t7	temps (caller-save)	31	ra	return addr (for jal)
16-23	s0-s7	saved (callee-save)			

*you don't need to memorize this!

(see kern/arch/mips/include/kern/regdefs.h).

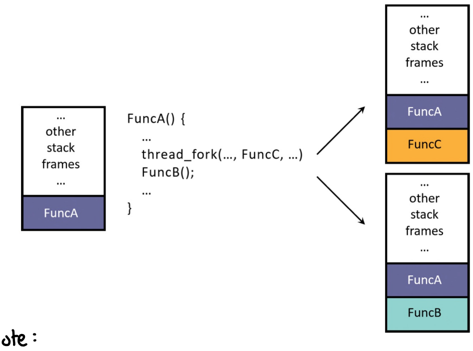
CONCURRENT PROGRAM EXECUTION (2 THREADS)

1. Each thread executes sequentially using its private register contents and data.



the code is shared between threads

2. The stack gets duplicated with thread_fork; when it is executed, a copy of the stack & register values are copied over into the new thread.



3. Note:
   ① Both threads are executing the same program.
   ② Both threads share the same code, read-only data, global variables & heap, but each thread may be executing different functions within the code.
   ③ Each thread has its own stack and program counter (PC).
   ④ Each thread has a fixed size.

IMPLEMENTING CONCURRENT THREADS: TIMESHARING

1. Options:
   ① Hardware support
      - P processors, C cores, M multithreading per core
      - PxCxM can execute simultaneously
   ② "Timesharing" — multiple threads take turns on the same hardware, rapidly switching between threads so they all make progress.
   ③ Both ① & ②!

MULTICORE PROCESSOR

1. A "multi-core processor" has the property that / a single die / computer chip can contain more than one processor unit.
2. These units can share some components, eg L3 cache, but execute code separately.

MULTITHREADING

1. "Multi-threading" refers to the property of having multiple threads run on the same core.
2. We need specialized hardware to do this.

CONTEXT SWITCH [IN TIMESHARING]

1. A "context switch" is the switch from one thread to another in time-sharing.
2. Process:
   ① "Scheduling" — CPU decides which thread to run next
   ② Save register contents of current thread
   ③ Load register contents of next thread
3. The "thread context" (register values) must be saved / restored carefully, since thread execution continuously changes the context.

CONTEXT SWITCHING ON MIPS

```
/* See kern/arch/mips/thread/switch.S */

switchframe_switch:
  /* a0: address of switchframe pointer of old thread. */
  /* a1: address of switchframe pointer of new thread. */

   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -40

   sw   ra, 36(sp)     /* Save the registers */
   sw   gp, 32(sp)
   sw   s8, 28(sp)     /* a.k.a. frame pointer */
   sw   s6, 24(sp)
   sw   s5, 20(sp)
   sw   s4, 16(sp)
   sw   s3, 12(sp)
   sw   s2, 8(sp)
   sw   s1, 4(sp)
   sw   s0, 0(sp)

   /* Store the old stack pointer in the old thread */
   sw   sp, 0(a0)


   /* Get the new stack pointer from the new thread */
   lw   sp, 0(a1)
   nop           /* delay slot for load */

   /* Now, restore the registers */
   lw   s0, 0(sp)
   lw   s1, 4(sp)
   lw   s2, 8(sp)
   lw   s3, 12(sp)
   lw   s4, 16(sp)
   lw   s5, 20(sp)
   lw   s6, 24(sp)
   lw   s8, 28(sp)        /* a.k.a. frame pointer */
   lw   gp, 32(sp)
   lw   ra, 36(sp)
   nop                       /* delay slot for load */

   /* and return. */
   j ra
   addi sp, sp, 40      /* in delay slot */
   .end switchframe_switch
```

→ to avoid load-use hazards.

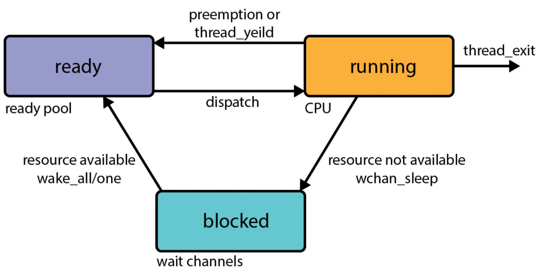
Switchframe_switch

1. The C function `thread_switch` calls the subroutine `switchframe_switch`.
2. `thread_switch` is the caller; it saves & restores the caller-save registers, including the return address (ra)
3. `switchframe_switch` is the callee; it must save & restore the callee-save registers, including the FP in OS/161.

* high-level functions are in C; low-level functions are in assembly.

THREAD STATES

1. States:
   ① "Running" — currently executing
   ② "Ready" — ready to execute
   ③ "Blocked" — waiting for something, so not ready to execute.



TIMESHARING

"Timesharing" is concurrency achieved by rapidly switching between threads.

SCHEDULING QUANTUM

The "scheduling quantum" is a limit on processor time that is an upper bound on how long a thread can run before it needs to yield to the processor.

PREEMPTION

"Preemption" forces a running thread to stop running, so another thread can run.
- to implement this, the thread library must have a means of "getting control".

This is usually accomplished using "interrupts".

INTERRUPTS

An "interrupt" is an event that occurs during a program's execution, and are caused by system devices (hardware).

When an interrupt occurs, the hardware automatically transfers control to a fixed location in memory.

Here, the thread library must place the "interrupt handler".

INTERRUPT HANDLER

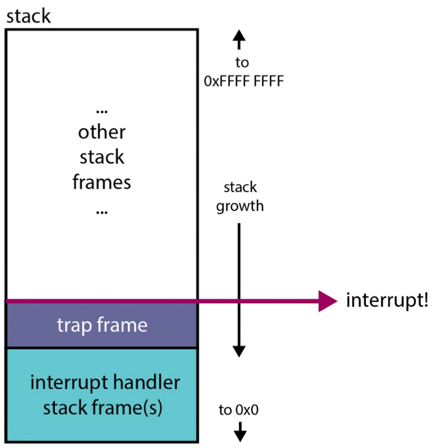
The "interrupt handler" is a procedure that:

① creates a "trap frame" to record the thread context at time of interrupt;

② determines which device caused the interrupt & performs device-specific processing;

③ restores the saved thread context from the trap frame & resumes execution of the thread.

OS/161 THREAD STACK AFTER AN INTERRUPT



SCHEDULING

"Scheduling" means deciding which thread should run next.

This is implemented by a "scheduler", which is a part of the thread library.

PREEMPTIVE ROUND-ROBIN SCHEDULING

"Pre-emptive round-robin scheduling" involves:

① scheduler maintains the "ready queue" of threads;

② on a context switch, the running thread is moved to the end of the ready queue, and the first thread on the queue is allowed to run.

③ Newly created threads are placed at the end of the ready queue.

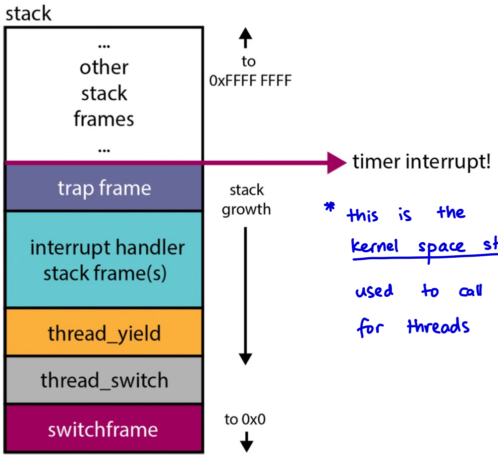
* OS/161 uses this type of scheduling.

The order threads run is non-deterministic since threads

① can be migrated to other cores; or

② interrupted by a device.

OS/161 THREAD STACK AFTER PRE-EMPTION



* this is the kernel space stack — used to call kernel functions for threads (eg `thread_yield`)

VOLUNTARY CONTEXT SWITCH

In a "voluntary context switch",

① The thread calls `thread_yield`, which calls `thread_switch`.

② This picks another runnable thread & calls `switchframe_switch` to store a switchframe on the stack.

③ We then do the context switch.

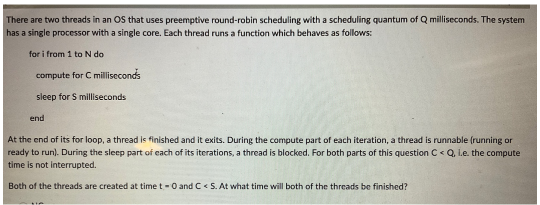
INVOLUNTARY CONTEXT SWITCH

In an "involuntary context switch":

① The thread is interrupted, resulting in a trap frame, followed by a stack frame for an interrupt handler.

② This calls thread_yield, and then we follow the steps for a voluntary context switch.

EXAMPLE

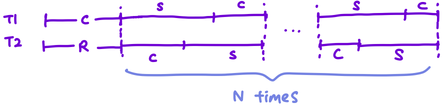


There are two threads in an OS that uses preemptive round-robin scheduling with a scheduling quantum of Q milliseconds. The system has a single processor with a single core. Each thread runs a function which behaves as follows:

```
for i from 1 to N do
  compute for C milliseconds
  sleep for S milliseconds
end
```

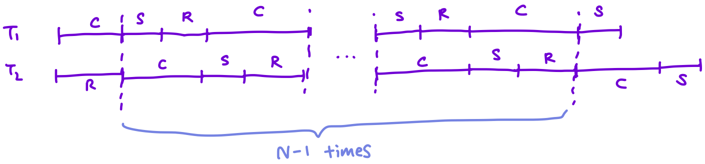
At the end of its for loop, a thread is finished and it exits. During the compute part of each iteration, a thread is runnable (running or ready to run). During the sleep part of each of its iterations, a thread is blocked. For both parts of this question C < Q, i.e. the compute time is not interrupted.

Both of the threads are created at time t = 0 and C < S. At what time will both of the threads be finished?



$\Rightarrow$ time $= C + N(C+S)$

If $C > S$:



$\Rightarrow$ time $= C + (N-1)(2C) + C + S = 2NC + S$

Topic 3: Synchronization

MOTIVATION

Consider

```
int volatile total = 0;

void add() {                    void sub() {
   int i;                          int i;
   for (i=0; i<N; i++) {           for (i=0; i<N; i++) {
      total++;                        total--;
   }                               }
}                               }
```

If one thread executes add & one executes sub, what is `total` when they have finished?

The value of `total` can not be 0!

To understand this, consider the code at the (pseudo) assembly language level.

TRACE 1: SEQUENTIAL EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9=0
add R9 1      ; R9=1
sw R9 0(R8)   ; total=1
          ----- Preemption ----->
                           loadaddr R10 total
                           lw R11 0(R10)   ; R11= 1
                           sub R11 1       ; R11= 0
                           sw R11 0(R10)   ; total= 0
```

→ total = 0.

TRACE 2: INTERLEAVED EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9= 0
add R9 1      ; R9= 1
          ----- Preemption ----->
                           loadaddr R10 total
                           lw R11 0(R10)   ; R11= 0
                           sub R11 1       ; R11= -1
                           sw R11 0(R10)   ; total= -1
         <----- Preemption -----
sw R9 0(R8) ; total= 1
```

→ total = 1

TRACE 3: MULTI-CORE EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9=0        loadaddr R10 total
add R9 1      ; R9=1        lw R11 0(R10)   ; R11=0
sw R9 0(R8)   ; total=1     sub R11 1       ; R11=-1
                            sw R11 0(R10)   ; total=-1
```

→ total = -1.

RACE CONDITION

A "race condition" is when the program result depends on the order of execution of the threads.

These occur when multiple threads are reading and writing the same memory at the same time.

Sources:

- implementation of code.

To identify the critical sections for race conditions:

① inspect each variable and ask if it is possible for multiple threads to read & write it concurrently.

② note constants & memory that are read-only by all threads do not cause race conditions.

ENFORCING MUTUAL EXCLUSION WITH LOCKS

```
int volatile total = 0;

void add() {                    void sub() {
   int i;                          int i;
   for (i=0; i<N; i++) {           for (i=0; i<N; i++) {
   ------------- start mutual exclusion -------------
        total++;                        total--;
   ------------- stop mutual exclusion  -------------
   }                                  }
}                               }
```

MUTEX / MUTUAL EXCLUSION

"Mutex" is a method to ensure only one thread can access this region at a time.

LOCK

A lock is a mechanism to provide mutual exclusion.

```
int volatile total = 0;
bool volatile total_lock = false;  // false means unlocked
                                   // true means locked

void add() {                       void sub() {
   int i;                             int i;
   for (i=0; i<N; i++) {              for (i=0; i<N; i++) {
   Acquire(&total_lock);               Acquire(&total_lock);
      total++;                            total--;
   Release(&total_lock);               Release(&total_lock);
   }                                   }
}                                  }
```

- acquire/release must ensure that only 1 thread can hold the lock at a time
- if a thread cannot acquire the lock immediately, it must wait until the lock becomes available

One possible implementation of acquire & release:

```
Acquire(bool *lock) {
  while (*lock == true){};  // spin until lock is free      *
  *lock = true;             // grab the lock
}

Release(bool *lock) {
  *lock = false;            // give up the lock
}
```

- "spinning" – a thread continually checks for a variable to change in a while loop.

But this does not work.

Why? → in the time a thread breaks out of the while loop (ie *) it could be preempted & another thread could then set *lock to true.

→ so there are 2 threads in the critical section.

TEST-AND-SET

Approach: create a function that given the address of a lock, `lock`, between when the `*lock` is tested & set, no other thread can change its value.

MIPS SYNCHRONIZATION INSTRUCTIONS

We use
① "ll" (load linked): load value at address addr
② "sc" (store conditional): store new value at addr if the value at addr has not changed since ll was executed.

2. sc returns "success" if the value stored at addr has not changed since ll was executed.

3. Implementation of locks:

```
MIPSTestAndSet(addr, new_val) {
   old_val = ll addr          // test
   status = sc addr, new_val  // set
   if ( status == SUCCEED ) return old_val
   else return true           // lock is being held
}

Acquire(bool *lock) {         // spin until hold lock
   while( MIPSTestAndSet(lock, true) == true ) {};
}
```

	lock value at ll	lock value before sc	lock value after sc	status	Lock State
1.	false	false	true	succeed	lock acquired
2.	true	true	true	succeed	keep spinning, no lock
3.	false	true	true	fail	keep spinning, no lock
4.	true	false	false	fail	keep spinning, no lock

SPINLOCKS IN OS/161

1. A "spin-lock" is a lock that spins; ie it repeatedly tests the lock's availability in a loop until the lock is returned.

2. When threads use a spin-lock, they "busy-wait" — they use the processor whilst they wait for the lock.

3. Spinlocks are efficient for short waiting times. (a few instructions).

4. OS/161 uses spinlocks to
① update shared variables / data structures
② implement other synchronization primitives (eg locks, semaphores, condition variables)

5. OS/161 uses ll & sc to implement test-and-set.

LOCKS IN OS/161

1. Locks also enforce mutex, but they block (instead of spinning, like spinlocks).

2. Locks can be used to protect larger critical sections without being a burden on the processor.

SPINLOCKS VS LOCKS

Similarities:
① The thread would `acquire` it (access the critical section) & `release` it (when it is done)
② Both need to be initialized / created before using them.
③ Both have `do_i_hold`.
④ Both have owners & can only be released by their owner. But:
- a spinlock is owned by a CPU;
- a lock is owned by a thread.

Why? → in OS/161, interrupts are disabled on the CPU that holds the spinlock but not other CPUs
→ to minimize spinning.

THREAD BLOCKING

When a thread blocks, it stops running; ie it stops using the processor.
① the scheduler chooses a new thread to run;
② a context switch from the blocking thread to the new thread occurs; &
③ the blocking thread is queued in a wait channel. (not the ready queue).
④ eventually, a blocked thread is signaled & awakened by another thread.

WAIT CHANNELS IN OS/161

1. "Wait channels" are used to implement thread blocking in OS/161.

- `void wchan_sleep(struct wchan *wc);`
  - blocks calling thread on wait channel `wc`
  - causes a context switch, like `thread_yield`
- `void wchan_wakeall(struct wchan *wc);`
  - unblock all threads sleeping on wait channel `wc`
- `void wchan_wakeone(struct wchan *wc);`
  - unblock one thread sleeping on wait channel `wc`
- `void wchan_lock(struct wchan *wc);`
  - prevent operations on wait channel `wc`
  - more on this later!

SEMAPHORES

"Semaphores" are synchronization primitives that can be used to enforce mutual exclusion requirements, & can be used to solve other kinds of synchronization problems.

It is an object with an integer value & supports 2 operations:

① "P" (procure): if the semaphore value > 0, decrement it; otherwise, wait until the value is > 0 & then decrement it.

② "V" (vacate): increment the value of the semaphore.

TYPES OF SEMAPHORES

"Binary semaphore" — a semaphore with a single resource; behaves like a lock but does not keep track of ownership.

"Counting semaphore": a semaphore with an arbitrary # of resources.

"Barrier semaphore": a semaphore used to force 1 thread to wait for others to complete; initial count is typically zero.

LOCK VS SEMAPHORE

Differences:

① V does not have to follow P;

② A semaphore can start with a count of 0 (ie no resources available).

③ Calling V increments the semaphore by 1;

④ This sequence of operations forces a thread to wait until resources are produced before continuing;

⑤ Semaphores do not have owners; locks do.

IMPLEMENTATION OF SEMAPHORES

```
P(struct semaphore * sem) {
  spinlock_acquire(&sem->sem_lock);
  while (sem->sem_count == 0) {        // test
      wchan_lock(sem->sem_wchan);      // prepare to sleep
      spinlock_release(&sem->sem_lock);
      wchan_sleep(sem->sem_wchan);     // sleep
      spinlock_acquire(&sem->sem_lock);
  }
  sem->sem_count--;                    // set
  spinlock_release(&sem->sem_lock);
}
```

```
V(struct semaphore * sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```

DESIGNING MULTITHREADED CODE

When designing/debugging multi-threaded code, consider what could go wrong if our code was preempted at each step at the procedure.

CONDITION VARIABLES

"Condition variables" have 3 operations.

① "Wait": this causes the calling thread to block, & releases the lock associated with the condition variable. Once the thread is unblocked it reacquires the lock.

② "Signal": if threads are blocked on the signaled condition variable, then one of those threads are unblocked.

③ "Broadcast": like signal, but unblocks all threads that are blocked on the condition variable.

Condition variables are intended to work together with locks.

These are also only used from within the critical section that is protected by the lock.

Condition variables allow threads to wait for arbitrary conditions to become true inside a critical section.

① If the condition is false, a thread can `wait` on the corresponding condition variable until it becomes true.

② If it is true, it uses `signal` or `broadcast` to notify any blocked threads.

EXAMPLE: GEESE

```
int volatile numberOfGeese = 100;
lock geeseMutex;
cv zeroGeese;   → condition variable

int SafeToWalk() {
    lock_acquire(geeseMutex);
    while (numberOfGeese > 0) {
        cv_wait(zeroGeese, geeseMutex);
    }
    GoToClass();
    lock_release(geeseMutex);
}
```

- `cv_wait` handles releasing/reacquiring the lock passed in.
- it will put the calling thread in the cv's wait channel to block
- cv_signal & cv_broadcast wake threads waiting on the cv.

VOLATILE

💡1 Race conditions can occur for reasons beyond the programmer's control; in particular they can be caused by

① the compiler; or

② the CPU.

💡2 These are due to optimizations to make the code run faster.

💡3 Compilers optimize for this difference, storing values in registers for as long as possible.

💡4 But for shared variables, this might not work.

eg

```
int sharedTotal = 0;
int FuncA() { ... code that modifies sharedTotal ... }
int FuncB() { ... code that modifies sharedTotal ... }
```

- if funcA stores sharedTotal in register S1
- & funcB stores it in register S2
- which register has the correct value?

💡5 "Volatile" disables this, forcing the value to be loaded from & stored to memory with each use.

- prevents compiler from re-ordering loads & stores for that variable.

💡6 Shared variables should be declared `volatile` !

DEADLOCKS

💡1 We say threads are "deadlocked" if neither can make progress, as they are blocking each other.

eg

```
lock lock1, lock2;
int FuncA() {                    int FuncB() {
    lock_acquire(lock1)              lock_acquire(lock2)
    lock_acquire(lock2)              lock_acquire(lock1)
    ...                             ...
    lock_release(lock1)              lock_release(lock2)
    lock_release(lock2)              lock_release(lock1)
}                                  }
```

What happens if the instructions are interleaved as follows:

- Thread 1 executes `lock_acquire(lock1)`
- Thread 2 executes `lock_acquire(lock2)`
- Thread 1 executes `lock_acquire(lock2)`
- Thread 2 executes `lock_acquire(lock1)`

This is what happens...

- Thread 1 executes `lock_acquire(lock1)` and holds it.
- Thread 2 executes `lock_acquire(lock2)` and holds it.
- Thread 1 executes `lock_acquire(lock2)` and blocks because `lock2` is being held by Thread 2.
- Thread 2 executes `lock_acquire(lock1)` and blocks because `lock1` is being held by Thread 1.

💡2 How to prevent deadlocks?

① "No Hold & Wait" : prevent a thread from requesting resources if it currently has resources allocated to it.

- a thread may hold several resources, but must obtain them all at once.

eg

```
lock_acquire(lock1);    // try get both locks
while(!try_acquire(lock2)) {
    lock_release(lock1); // didn't get lock2 so try again
    lock_acquire(lock1);
}                             // can now obtain both resources
```

② "Resource Ordering" : order the resource types.

- each thread must acquire resources in increasing resource type order.

Topic 4: Processes and the Kernel

PROCESS

💡1 A "process" is an environment in which an application program runs.

ie a "program that is running".

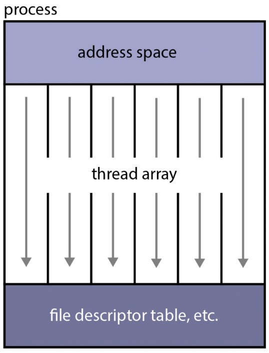
💡2 The process's environment includes virtualized resources that its program can use:

① Threads to execute code on processors;

② An "address space" for code & data;

③ A file system;

④ I/O.
eg keyboard, display



💡3 Processes are created & managed by the kernel, & each program's process isolates it from other programs running in other processes.

- when a program is executed, it seems like it has exclusive access to the processor, RAM & I/O even though they are actually shared.

VIRTUAL MEMORY

💡1 "Virtual memory" is an address space that employs virtualization with RAM & secondary storage.

💡2 It appears to have a large amount of primary memory (RAM) that the process has exclusive access to for its code, data & stacks.

VIRTUALIZED RESOURCE

💡1 A "virtualized resource" is to take a physical resource (eg hard drive) & create a more abstract version of it (eg file system).

PROCESS MANAGEMENT CALLS

💡1 Processes can be created, managed & destroyed, & each OS supports a variety of functions to support these tasks.

	Linux	OS/161
Creation	fork,execv	fork,execv
Destruction	_exit,kill	_exit
Synchronization	wait, waitpid, pause, ...	waitpid
Attribute Mgmt	getpid, getuid, nice, getrusage, ...	getpid

fork

💡1 The system call `fork` creates a new process, the "child" (C), that is a clone of the parent (P), the original process.

- it copies P's address space (code, globals, heap, stack) into C's.
- it makes a new thread for C.
- it copies P's trap frame to C's (kernel) stack

💡2 The address space of P & C are identical at the time of the fork, but may diverge afterwards.

💡3 `fork` is called by the parent, but the function returns in both the parent & child.

- it returns 0 to the child process
- it returns the child's pid to the parent process
- this allows us to distinguish the child & parent.

_exit

💡 `_exit` terminates the process that calls it.

- a process can supply an exit status code when it exits
- the kernel records the exit status code in case another process needs it, via waitpid.

waitpid

💡 `waitpid` lets a process wait for another to terminate/block, and then retrieves its exit status code.

* pid = process identifier

EXAMPLE 1 : fork, _exit, getpid, waitpid

```
main() {
    int rc = fork();    // returns 0 to child, pid to parent
    if (rc == 0) {      // child executes this code
      my_pid = getpid();
      x = child_code();
      _exit(x); ←—— ends process w/ return value x
    } else {            // parent executes this code
      child_pid = rc;
      parent_pid = getpid();
      parent_code();
      p = waitpid(child_pid,&child_exit,0); ← wait for child to exit. get its exit status
      if (WIFEXITED(child_exit)) ←— returns true if child called _exit()
        printf("child exit status was %d",
          WEXITSTATUS(child_exit)) ←— returns exit status code
    }
}
```

getpid

💡 `getpid` returns the process identifier (ie pid) of the current process.

- each existing process has its own unique pid to identify it

execv

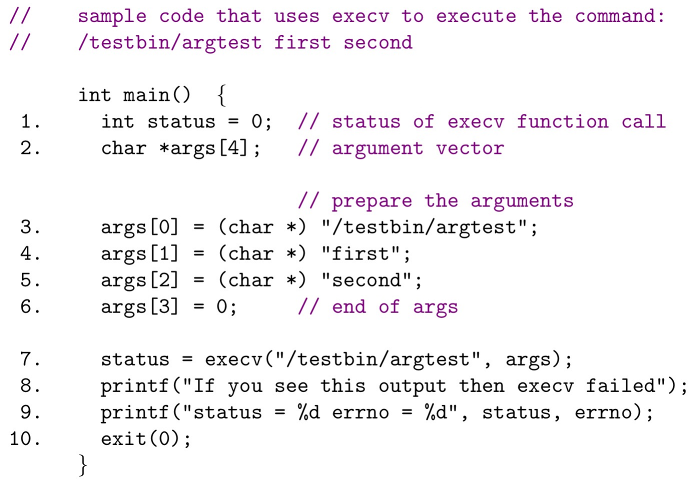
💡1 `execv` changes the program that a process is running.

- the calling process's current virtual memory is destroyed
- the process gets a new virtual memory (code, heap, stack etc) initialized with the code & data of the new program to run.
- the pid remains the same.

💡2 After `execv`, the new program starts executing.

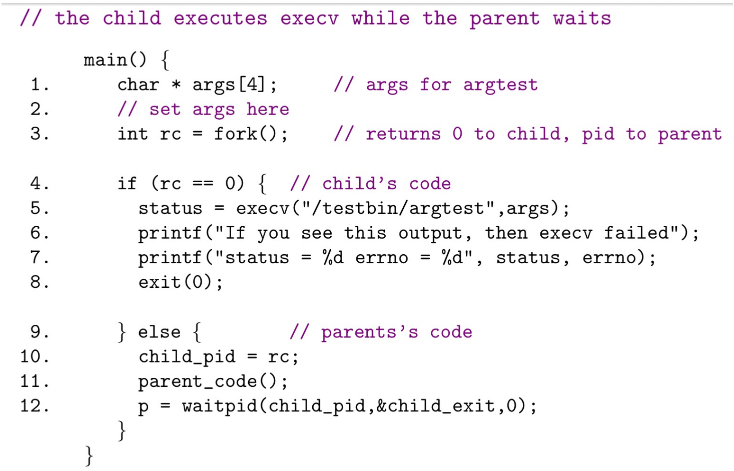
💡3 `execv` can pass arguments to the new program if required.

EXAMPLE 2 : execv



```
//    sample code that uses execv to execute the command:
//    /testbin/argtest first second

      int main()  {
        int status = 0;  // status of execv function call
        char *args[4];   // argument vector

                         // prepare the arguments
        args[0] = (char *) "/testbin/argtest";
        args[1] = (char *) "first";
        args[2] = (char *) "second";
        args[3] = 0;     // end of args

        status = execv("/testbin/argtest", args);
        printf("If you see this output then execv failed");
        printf("status = %d errno = %d", status, errno);
        exit(0);
      }
```

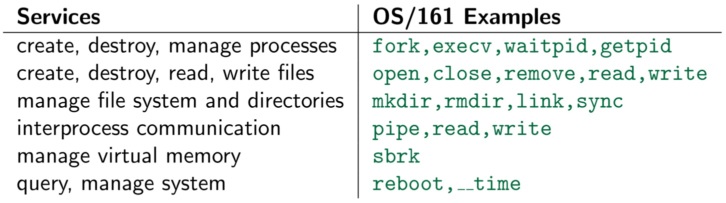
EXAMPLE 3 : execv & fork



```
// the child executes execv while the parent waits

     main() {
        char * args[4];     // args for argtest
        // set args here
        int rc = fork();    // returns 0 to child, pid to parent

        if (rc == 0) {  // child's code
          status = execv("/testbin/argtest",args);
          printf("If you see this output, then execv failed");
          printf("status = %d errno = %d", status, errno);
          exit(0);

        } else {         // parents's code
          child_pid = rc;
          parent_code();
          p = waitpid(child_pid,&child_exit,0);
        }
     }
```

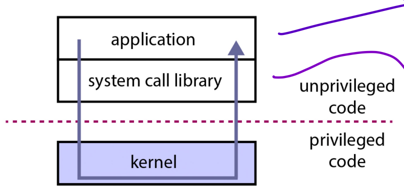
SYSTEM CALLS

- "System calls" are the interface between user processes & the kernel.



Services	OS/161 Examples
create, destroy, manage processes	fork,execv,waitpid,getpid
create, destroy, read, write files	open,close,remove,read,write
manage file system and directories	mkdir,rmdir,link,sync
interprocess communication	pipe,read,write
manage virtual memory	sbrk
query, manage system	reboot,__time

SOFTWARE STACK



programs from previous courses (eg CS 241/246)

library routines supplied by the programming language for using OS services.

EXECUTION PRIVILEGE

1. The processor implements different levels of "execution privilege" as a security & isolation mechanism.
2. The kernel code runs at the highest privilege level.
3. Applications run at a lower privilege since user programs should not be permitted to perform certain tasks.
   - eg modifying page tables, halting the processor.

INTERRUPTS

1. "Interrupts" are generated by devices when they need attention.
2. These cause the processor to transfer control to a fixed location in memory, where an "interrupt handler" is located.
   - this is part of the kernel.
   - if an interrupt occurs, control will jump from the app to the kernel's interrupt handler routine.

EXCEPTIONS

1. "Exceptions" are caused by instruction execution when a running program needs attention.
   eg arithmetic overflow, illegal instructions/addresses
2. When these occur, control is transferred to a fixed location where an exception handler is located.
3. The processor then switches to privileged execution code.
4. In OS/161, the same routine is used to handle exceptions & interrupts.

MIPS EXCEPTION TYPES

```
EX_IRQ    0    // Interrupt
EX_MOD    1    // TLB Modify (write to read-only page)
EX_TLBL   2    // TLB miss on load
EX_TLBS   3    // TLB miss on store
EX_ADEL   4    // Address error on load
EX_ADES   5    // Address error on store
EX_IBE    6    // Bus error on instruction fetch
EX_DBE    7    // Bus error on data load *or* store
EX_SYS    8    // Syscall
EX_BP     9    // Breakpoint
EX_RI     10   // Reserved (illegal) instruction
EX_CPU    11   // Coprocessor unusable
EX_OVF    12   // Arithmetic overflow
```

PERFORMING A SYSTEM CALL

1. To perform a system call, the application program needs to cause an exception to make the kernel execute.
   - exception code: EX_SYS
   - thrown using the command `syscall` on MIPS
2. The kernel's exception handler checks the exception code to distinguish system call exceptions from other types.

WHICH SYSTEM CALL? (SYSTEM CALL CODES)

1. The kernel defines a code for each system call it understands.
   - since there is only one syscall exception.
2. The kernel expects the app to place the appropriate system call code in a specified location before executing `syscall`.
3. The codes & code location are part of the kernel's "ABI" (application binary interface).

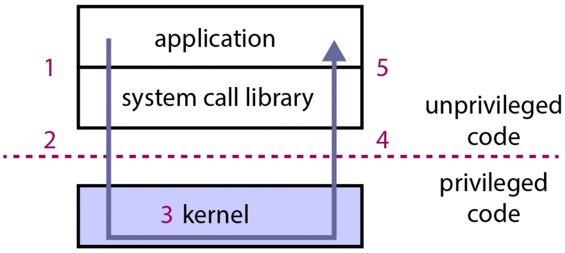
SOME OS/161 SYSTEM CALL CODES

```
#define SYS_fork          0
#define SYS_vfork         1
#define SYS_execv         2
#define SYS__exit         3
#define SYS_waitpid       4
#define SYS_getpid        5
```

SYSTEM CALL PARAMETERS

To pass parameters to system calls, the application places the arguments in kernel-specified locations before the `syscall`, and receives the return values in kernel-specified locations after the exception handler returns.

SYSTEM CALL SOFTWARE STACK REVISITED



1. The application calls a library wrapper function for the system call.
2. The library function executes a `syscall` instruction.
3. This causes the kernel's exception handler to run, which
   ① creates trap frame to save application program state
   ② determines this is a system call exception
   ③ determines what system call is being requested
   ④ does the work for the requested system call
   ⑤ restores the application program state from the trap frame; &
   ⑥ returns from the exception.
4. Then, the library wrapper function finishes & returns.
5. The application can finally continue execution.

USER VS KERNEL STACK

1. The "user stack" is used while the thread is executing application code.
   - this holds stack frames for the application
   - kernel creates this stack when it sets up the virtual address space for the process
2. The "kernel stack" is used while the thread is executing kernel code.
   - ie after an exception/interrupt
   - in OS/161, the `t-stack` field of a thread points to this
   - this holds stack frames for kernel functions
   - and holds trap frames & switch frames.

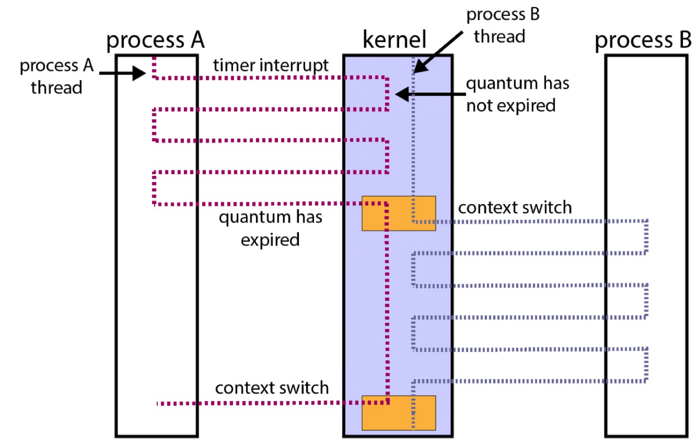
MULTIPROCESSING / MULTITASKING

"Multiprocessing" means having multiple processes existing at the same time.

All processes must share the available hardware resources, and the sharing is coordinated by the OS.

The OS ensures that processes are isolated from one another.

TWO PROCESS EXAMPLE



INTERPROCESS COMMUNICATION (IPC)

IPC is a family of methods used to send data between (otherwise isolated) processes.

Methods:

① "File" - data to be shared is written to a file, accessed by both processes.

② "Socket" – data is sent via network interface between processes.

③ "Pipe" — data is sent, unidirectionally, from one process to another via a OS-managed data buffer.

④ "Shared memory" – data is sent via block of shared memory visible to both processes.

⑤ "Message passing/queue" – a queue / data stream provided by the OS to send data between processes.

Topic 5: Virtual Memory

1. Motivation: How can we manage primary memory (ie RAM) so that
   ① multiple processes can be loaded into RAM;
   ② a bug in a program that is executed in one process cannot affect another process; &
   ③ the implementation details are mostly hidden?
2. Idea: Create a virtual environment for RAM (ie "virtual memory") that looks the same for all processes.
3. We know virtual memory exists as incrementing one process's variables does not affect the other's variables, although the memory locations of the code & global variables are the same across both variables.

PHYSICAL ADDRESSES

1. "Physical addresses" are provided directly by the hardware; ie the amount of installed RAM.
2. There is one physical address space per computer.

VIRTUAL ADDRESSES

1. "Virtual addresses" are provided by the OS to the processes.
2. There is one virtual address space per process.

ADDRESS TRANSLATION

"Address translation" is the process of converting virtual addresses to physical addresses (via the hardware & OS).

WHY VIRTUAL MEMORY?

Virtual memory provides each process with the illusion that it has a large amount of contiguous memory to itself (the "address space").

PHYSICAL MEMORY

1. If it takes P bits to specify the physical address of each byte, then the max amount of addressable physical memory is 2^P bytes.
2. Notes:
   ① P=32 in Sys/161.
   ② P=18 in the lecture notes.

VIRTUAL MEMORY

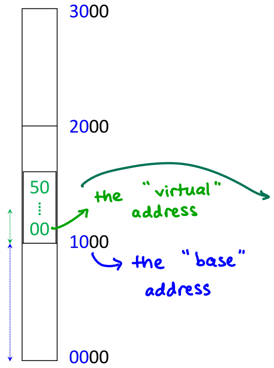
1. If it takes V bits to specify the virtual address of each byte, then the max amount of addressable virtual memory is 2^V bytes.
2. Running applications only see virtual addresses.
3. Each process is isolated in its virtual memory & cannot access another process's virtual memory.

MEMORY MANAGEMENT UNIT (MMU)

The MMU is the part of the hardware that performs address translation using information provided by the kernel.

ADDRESS TRANSLATION: BASIC IDEA

1. Each virtual memory address is mapped to a different part of physical memory.
2. Idea:



* Then, to convert from a virtual addr to physical addr, add the base address to the virtual address.

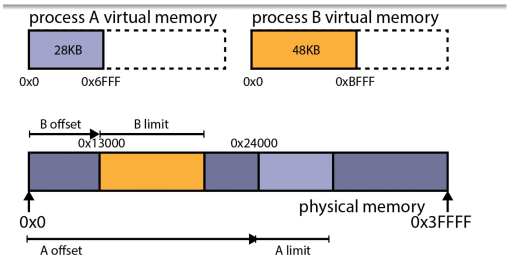
⇒ The virtual address of this is 1000 + 00 – 1000 + 50

⇒ $\underbrace{1000 - 1050}$.

↳ the "physical" address.

ADDRESS TRANSLATION METHOD 1: DYNAMIC RELOCATION

1 Idea:



The virtual address of each process is translated using 2 values:

① "offset", which is stored in a "relocation register" (`offset`)
— the address in physical memory where the process's memory begins.

② "limit", stored in the `limit` register
— the amount of memory used by the process.

2 Converting a virtual addr v to a physical address p, the MMU does:

```
if (v < limit) then
    p ← v + offset
else raise memory exception
```

3 The kernel maintains separate offset & limit values for each process, & changes these values in the registers during a context switch between processes.

PROPERTY 1: FRAGMENTATION OF PHYSICAL MEMORY

1 The OS must track which parts of physical memory are free & which are in-use.

2 So, the OS must allocate / deallocate variable-sized chunks of physical memory.

3 This potentially causes fragmentation of physical memory, where we ideally want it to be contiguous.

PROPERTY 2: ISOLATION OF PROCESSES

Dynamic relocation
① Allows multiple processes to share RAM; &
② isolates each process's address space from all other processes

using just 2 registers per process.

EXAMPLE 1

	Process A		Process B	
	Limit Register: 0x0000 7000		Limit Register: 0x0000 C000	
	Relocation Register: 0x0002 4000		Relocation Register: 0x0001 3000	
①	v = 0x0000	p = 2 4000	v = 0x0000	p = ?
	v = 0x102C	p = 2 502C	v = 0x102C	p = ?
②	v = 0x8000	p = exception	v = 0x8000	p = ?

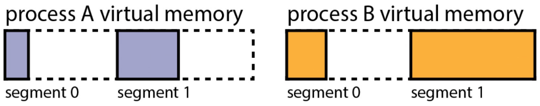
① p = v + limit
= 0x 0002 4000 + 0 x 0000 0000
= 0x 24000

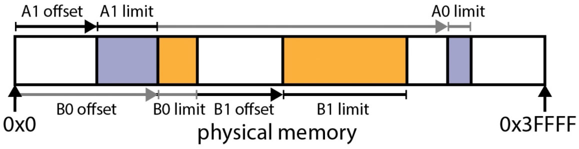
② v > limit so exception!

ADDRESS TRANSLATION METHOD 2: SEGMENTATION

💡1 Motivation: Virtual memory may be large, but the process's address space might be very small & discontiguous.

💡2 Idea: Instead of mapping the entire virtual memory to physical memory, we map each segment of the address space separately.





💡3 The kernel maintains an offset & limit value for each segment.

💡4 With segmentation, a virtual address can be thought of as having two parts:

① a segment ID; &

② the offset within segment.

💡5 If we have k bits for the segment ID, we have up to

- 2^k segments; &
- 2^{v-k} bytes per segment.

ADDRESS TRANSLATION METHOD 1

💡1 For each segment i, the MMU has

① a relocation offset register: offset[i]

② a limit register: limit[i].

💡2 Translating a virtual address v → physical addr p:

```
s ← SegmentNumber(v)
a ← OffsetWithinSegment(v)
if  (a ≥ limit[s])  then
    raise memory exception
else
    p ← a + offset[s]
```

EXAMPLE 1

Process A

Segment	Limit Register	Relocation Register
0	0x2000	0x38000
1	0x5000	0x10000

Process B

Segment	Limit Register	Relocation Register
0	0x3000	0x15000
1	0xB000	0x22000

Translate the following for process A and B: (we do A here)

v	Segment	Offset	p
0x1240	0	0x1240	0x39240
0xA0A0	1	0x20A0	0x120A0
0x66AC	0	0x66AC	exception
0xE880	1	0x6880	exception

First, convert the first hex into binary:

$$0x1240 \rightarrow [\underbrace{0}_{\text{segment \#}}\ \underbrace{0\,0\,1}_{\text{the first hexdigit of the offset}}]_2[\underbrace{2\ 4\ 0}_{\text{the rest of the offset.}}]_{16}$$

so

$$0x1240 \rightarrow \text{segment \# } = 0, \quad \text{offset} = 0x\underbrace{1}_{\text{"001"}}240$$

$$\therefore\ v = \text{relo}[\#] + \text{offset}$$

$$= \underbrace{0x38000}_{\text{the relocation register for segment 0}} + 0x1240 = 0x39240.$$

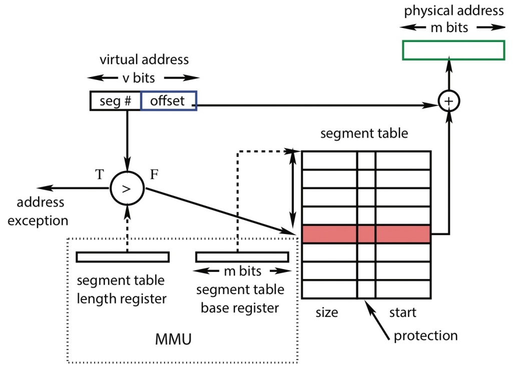
Another example:

$$0xE880 \rightarrow [\underbrace{1}_{\text{seg \#}}\ \underbrace{1\,1\,0}_{\text{first hex digit of offset}}]_2\ [\underbrace{8\ 8\ 0}_{\text{rest of offset.}}]_{16}$$

$\therefore$ seg # = 1, offset = 0x6880.

But limit[1] = 0x5000 < 0x6880 = offset! So we get an exception.

ADDRESS TRANSLATION METHOD 2: SEGMENT TABLE



💡 Idea:

① If segment # in v > # of segments, raise an exception.

② Otherwise, use the segment # to lookup the limit & relocation values from the segment table.

*see next page for example.

EXAMPLE 2

Virtual addr = 32 bits, Physical addr = 32 bits, Offset = 28 bits

```
Segment Table base reg   0x0010 0000
Segment Table len reg    0x0000 0004
```

Seg	Size	Prot	Start
0	0x6000	X-	0x7 0000
1	0x1000	-W	0x6 0000
2	0xC000	-W	0x5 0000
3	0xB000	-W	0x8 0000

```
Virtual address          0x0000 2004
Physical address  =      ___________ ?
Virtual address          0x2000 31A4
Physical address  =      ___________ ?
```

Sol

Solⁿ. ① Split virtual addr into segment # & offset.

eg $0x\,\underbrace{0}_{\text{seg \#}}\underbrace{0002004}_{\text{offset.}} \longrightarrow$ seg # = 0, offset = 0x 0002004

② Check segment # is not too large.

eg 0 < 0x4

③ Check the offset is not too large.

eg 0x2004 < 0x6000

④ Translate the segment number into a physical base address.

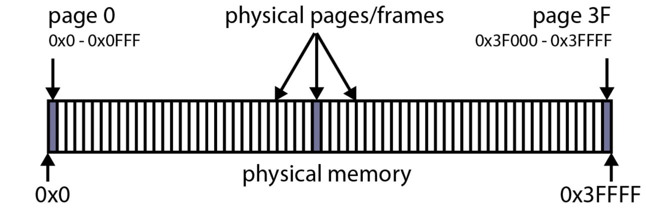
eg SegmentTable [0x0] = 0x0007 0000.

⑤ Add the offset to physical base address to get the physical address.

eg 0x0007 0000 + 0x0000 2004 = 0x0007 2004.

ADDRESS TRANSLATION METHOD 3: PAGING

💡 **Idea:** Divide physical memory into fixed-size chunks called "frames" / "physical pages".



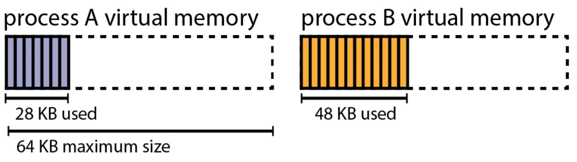
Here:
- physical addresses are 18 bits
  - ⇒ size of physical memory is $2^{18} = 256$ KB.
- frame size $= 2^{12} = 4$ KB; so
  - ⇒ physical memory consists of $2^{18}/2^{12} = 64$ (0x40) frames.

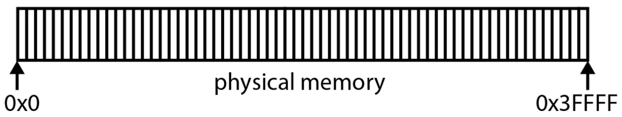
VIRTUAL MEMORY

💡$_1$ Idea:

① Divide virtual memory into fixed-sized chunks called "pages".

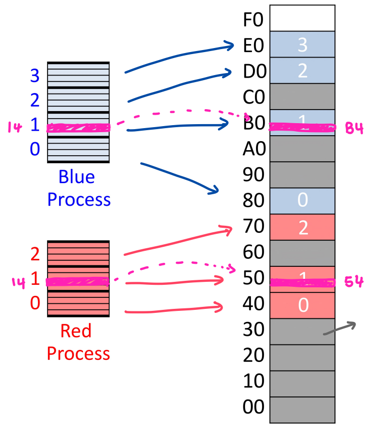
② $\underbrace{\text{Page size}}_{\text{virtual mem}} = \underbrace{\text{frame size}}_{\text{physical mem}}$





- virtual addr are 16 bits
- so a process can use up to $\frac{2^{16}}{2^{12}} = 16$ pages.

💡$_2$ How is it implemented?



The page table of the red process would be

VPN	FN
2	7
1	5
0	4

virtual page #

frame number

since virtual page 2 is mapped to address 70.

For the blue process:

VPN	FN
3	E
2	D
1	B
0	8

these are already used by other processes/frames

For the red process, virtual addr 14 → physical addr 54.

For the blue process, virtual addr 14 → physical addr 84.

PAGE TABLES

💡$_1$ Each process has its own page table, which maps pages (from virtual memory) to frames (in physical memory).

Process A Page Table

Page	Frame	Valid?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xF	0x00	0

Process B Page Table

Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
0x3	0x23	1
...	...	...
0xA	0x33	1
0xB	0x2C	1
0xC	0x00	0
0xD	0x00	0
0xE	0x00	0
0xF	0x00	0

💡$_2$ Each row of a page table is a "page table entry" (PTE).

💡$_3$ The "valid bit" indicates if the PTE is used or not.
- If it is 1, it maps the page to physical memory.
- If it is 0, it does not correspond to a page in virtual memory.

💡$_4$ $\boxed{\text{\# of PTEs = max virtual memory size / page size.}}$

ADDRESS TRANSLATION

💡$_1$ The MMU includes a page table base register which points to the page table for the current process.

💡$_2$ To translate a virtual address, the MMU:

① Determines the page # & offset of the virtual address;
- page # = virtual address / page size
- offset = virtual address % page size

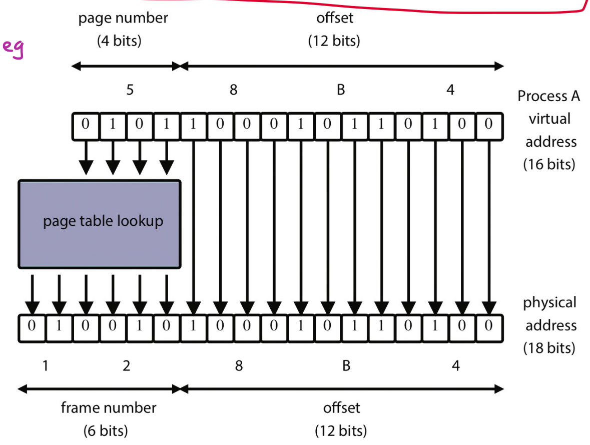
② Look up the PTE in the current process's PT using the page number to get the frame number.

③ Check the PTE is valid. If it isn't, raise an exception.

④ Otherwise,

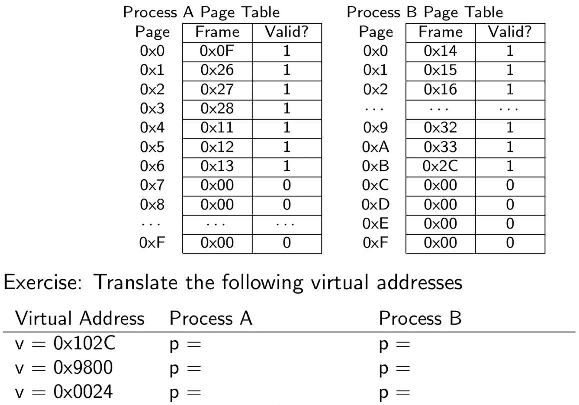
$\boxed{\text{physical addr} = (\text{frame \#} \times \text{frame size}) + \text{offset}.}$

eg



- # of bits for offset $= \log_2(\text{page size}) = 12$
- # of PTEs = max VM size / page size
- # of bits for page # $= \log_2(\text{\# of PTEs})$

EXAMPLE 1



Process A Page Table

Page	Frame	Valid?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
· · ·	· · ·	· · ·
0xF	0x00	0

Process B Page Table

Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
· · ·	· · ·	· · ·
0x9	0x32	1
0xA	0x33	1
0xB	0x2C	1
0xC	0x00	0
0xD	0x00	0
0xE	0x00	0
0xF	0x00	0

Exercise: Translate the following virtual addresses

Virtual Address	Process A	Process B
v = 0x102C	p =	p =
v = 0x9800	p =	p =
v = 0x0024	p =	p =

① Determine page # & offset.

- page # = 1 hex digit, offset = 3 hex digits
- 0x 102C → $\underbrace{1}_{\text{page \#}}\ \underbrace{02C}_{\text{offset}}$ (in A)

② Look up PTE using the current process PT.

- for A, PTE[1] is frame = 26, valid = true.

③ If PTE is invalid, raise an exception.

④ Otherwise, combine corresponding frame # from the PTE with the offset.

→ 0x $\underbrace{26}_{\text{frame \#}}\underbrace{02C}_{\text{offset}}$.

OTHER INFO FOUND IN PTES

PTEs may contain other fields.

Examples:

① Write protection bit
- to indicate page is read-only

② Bits to track page usage
- reference/use bit: has the process used this page recently?
- dirty bit: has the contents of the page been changed?

PAGE TABLE SIZE

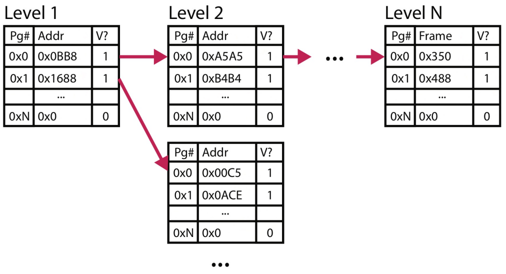
page table size = # of pages x size of a PTE

# of pages = max VM size / page size

☆ cutoff for midterm.

ADDRESS TRANSLATION METHOD 4: TWO-LEVEL PAGING

Idea: Instead of having a single PT to map an entire virtual memory, we can organize it and split the PT into multiple levels.



Pg#	Addr	V?
0x0	0x0BB8	1
0x1	0x1688	1
	...	
0xN	0x0	0

Pg#	Addr	V?
0x0	0xA5A5	1
0x1	0xB4B4	1
	...	
0xN	0x0	0

Pg#	Addr	V?
0x0	0x00C5	1
0x1	0x0ACE	1
	...	
0xN	0x0	0

Pg#	Frame	V?
0x0	0x350	1
0x1	0x488	1
	...	
0xN	0x0	0

- a large, contiguous table is replaced with smaller tables
- if a table contains no valid PTEs, do not create the table.

Example:

Single-Level Paging

Page	Frame	V?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	NULL	0
0x8	NULL	0
...	...	...
0xE	NULL	0
0xF	NULL	0

Two-Level Paging

Directory

Page	Address	V?
0x0	Table 1	1
0x1	Table 2	1
0x2	NULL	0
0x3	NULL	0

Table 1

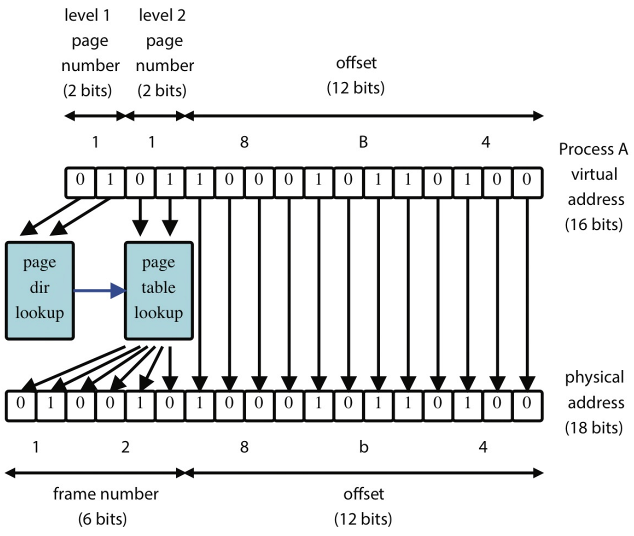
Page	Frame	V?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1

Table 2

Page	Frame	V?
0x0	0x11	1
0x1	0x12	1
0x2	0x13	1
0x3	NULL	0

The address translation is the same, but the lookup is now different.

ADDRESS TRANSLATION



Method:

① Split the virtual addr v into $v = p_1 \, p_2$ offset, where

eg $p_1 = 01$, $p_2 = 01$, offset = 8B4.

② Use p_1 to find the PT.

eg pagedir$[p_1]$ = pagedir[01] = Table 2

③ Use p_2 to index into the given PT.

eg Table2$[p_2]$ = Table2[01] = 0x12.

This is the frame number.

④ Combine the frame number with the offset to get the physical address.

↳ $0x\underbrace{12}_{\text{frame \#}}\underbrace{8B4}_{\text{offset.}}$

ADDRESS TRANSLATION METHOD 5: MULTI-LEVEL PAGING

1 Each virtual address has n+1 parts, $(p_1, \ldots, p_n, o)$.

2 Method:

① For $i = 1, \ldots, n$, index into the i^{th} page table directory using p_i to get a pointer to the $(i+1)^{th}$ level page table directory.

If $i = n$, the pointer gets the PTE of the page being accessed.

② For each step, if the entry is invalid, raise an exception.

③ Otherwise, combine the frame # with the offset o to determine the physical address.

3 Multi-level paging reduces the size of individual PTs.

4 In particular, we have each table fit on a single page.

5 Note

$$\# \text{ of pages (\& PTEs) needed} = \frac{\text{virtual memory size}}{\text{page size}}$$

$$\# \text{ of PTEs that can be stored on each page} = \frac{\text{page size}}{\text{PTE size}}$$

$$\# \text{ of PTs} = \frac{\# \text{ of PTEs}}{\text{PTE's per page}}$$

Then

$$\text{space needed for directory (top level page table)} = \# \text{ of PTs} \times \text{PTE size}.$$

6 When # of entries required in the directory is larger than the page size, add more levels to map virtual memories efficiently.

TRANSLATION LOOKASIDE BUFFER (TLB)

Motivation: Address translation through a PT adds a min of one extra memory operation (for PTE lookup) for each memory operation performed during instruction execution.

But this can be slow!

Idea: include a "TLB" (a cache for PTEs) in the MMU that

① acts as a small, fast, dedicated cache of address translations

② each TLB entry stores a single page # → frame # mapping.

HARDWARE-MANAGED TLB

Address translation:

```
if (p has an entry (p,f) in the TLB) then
   return f      // TLB hit!
else
    find p's frame number (f) from the page table
    add (p,f) to the TLB, evicting another entry if full
   return f      // TLB miss
```

If the MMU cannot distinguish TLB entries from different address spaces, the kernel must clear/invalidate the TLB on each context switch from one process to another.

Since the MMU handles TLB misses, PT lookup & replacement of TLB entries, the MMU must understand the kernel's PT format.

SOFTWARE-MANAGED TLB

Address translation:

(given v on page p)

```
if (p has an entry (p,f) in the TLB) then
   return f          // TLB hit!
else
   raise exception  // TLB miss
```

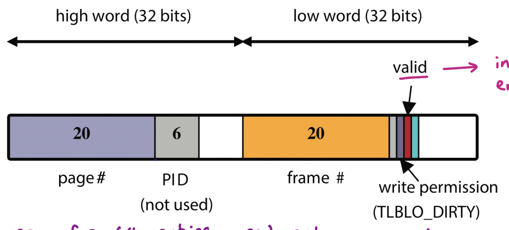
* MIPs uses a software-managed TLB

If TLB miss, then the kernel needs to

① determine the frame # for p

② add (p,f) to the TLB, evicting another entry if necessary.

MIPS R3000 TLB



→ indicates if TLB entry is valid

- room for 64 entries, each entry is 64 bits long
- if TLBLO_DIRTY (write permission) is set (=1), then you can write to the page.

* `pid` can be used to distinguish mappings from different processes, but OS/161 does not use it.

Note: in OS/161,

① virtual & physical addresses are 32 bits

② page size is 4KB (requiring 12 bits).

SUMMARY

Benefits:

① Paging does not cause external fragmentation.

② Multi-level paging reduces the amount of memory required to store page→frame mappings.

Costs:

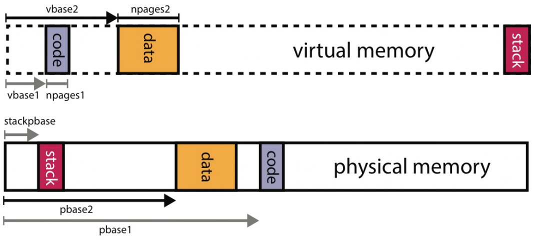
① TLB misses are increasingly expensive with deeper page tables.

VM IN OS/161 ON MIPS: dumbvm

1. In MIPS, TLB exceptions are handled by a kernel function `vm_fault`.
   - since MIPS uses a software-managed TLBs
   - so it raises an exception in a TLB miss.
2. `vm-fault` uses information from a kernel `addrspace` structure to determine a page→frame mapping to load into the TLB.
3. There is a separate `addrspace` structure for each process, which describe where its process' pages are stored in physical memory.
4. `addrspace` does the same role as a PT, but is simpler since OS/161 places all pages of each segment contiguously in physical memory.

addrspace

```
struct addrspace {
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
};
```



ADDRESS TRANSLATION

Method:

① Calculate the top & bottom address for each segment:

```
vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;
```

Where the following are predefined constants

```
USERSTACK = 0x8000 0000
DUMBVM_STACKPAGES = 0xC        // decimal 12
PAGE_SIZE = 0x1000             // decimal 4096 or 4K
```

which define the top, bottom and size of the stack in virtual memory.

② Then calculate the physical address:

```
if (faultaddress >= vbase1 && faultaddress < vtop1)  // check if faultaddress is in
    paddr = (faultaddress - vbase1) + as->as_pbase1;    the code segment

else if (faultaddress >= vbase2 && faultaddress < vtop2)  // check if faultaddress is in
    paddr = (faultaddress - vbase2) + as->as_pbase2;        the data segment

else if (faultaddress >= stackbase && faultaddress < stacktop)  // check if faultaddress is in
    paddr = (faultaddress - stackbase) + as->as_stackpbase;       the stack
             offset from virtual base addr   physical base addr
else
    return EFAULT;
```

EXAMPLE 1

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

	Process 1	Process 2
Virtual addr	0x0040 0004	0x0040 0004
Physical addr =	0x 200 004 ?	___________ ?
Virtual addr	0x1000 91A4	0x1000 91A4
Physical addr =	0x8091A4 ?	___________ ?
Virtual addr	0x7FFF 41A4	0x7FFF 41A4
Physical addr =	0x1001A4 ?	___________ ?
Virtual addr	0x7FFF 32B0	0x2000 41BC
Physical addr =	EFAULT ?	___________ ?

① Check if p belongs to any addresses:

vbase1 = 0x400000

vtop1 = vbase1 + as→ as_npages1 * PAGE_SIZE

= 0x400000 + 8 * 0x1000

= 0x 408000.

Since 0x400000 ≤ 0x400004 ≤ 0x 408000, p is in the code segment.

② If so, translate it.

paddr = (faultaddress − vbase1) + as→ as_pbase1

= (0x400004 − 0x400000) + 0x200000

= 0x 200004.

③ Otherwise, throw EFAULT.

INITIALIZING AN ADDRSPACE

1 When the kernel creates a process to run a particular program,

① it must create an addrspace for the process;
&
② load the programs code & data into the addrspace.

So, we are "pre-loading" the addrspace.

2 Other OSs load pages "on-demand", since

① the program loads faster; &
② it will not use up physical memory for unused features.

OBJECT / EXECUTABLE FILE & ELF

1 A program's code & data is described in an executable file.

2 OS/161 expect executable files to be in "ELF" (executable & linking format) format.

ELF FILES

1 ELF files contain address space segment descriptions.

2 The ELF header describes the segment's "images":

① The vaddr of the start of the segment
② The length of the segment in the vaddr-space
③ The location of the segment in the ELF file
④ The length of the segment in the ELF file

3 It also identifies the "entry point" — the vaddr of the programs first instruction.

4 It also contains other information (eg relocation tables) useful to compilers/linkers/etc.

OS/161 ELF FILES

1 In OS/161, the ELF file contains

① the "text segment", containing the program code & read-only data
② the "data segment", containing any other global program data

2 These are an exact copy of the binary data to be stored in the addrspace.

3 We create the stack segment for each process.

- the ELF file does not describe the stack, since the initial contents of the stack are unknown until run-time.

VIRTUAL MEMORY FOR THE KERNEL

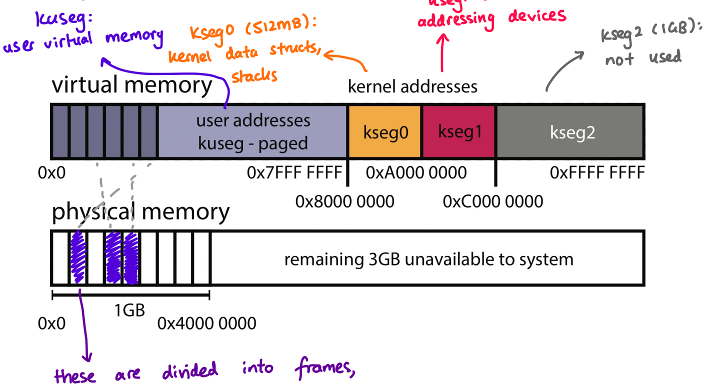
We want the kernel to live in virtual memory. Challenges:

① "bootstrapping": how can the kernel run in vm when it is starting since it helps to implement vm?

② "sharing": data might be copied between kernel & app programs; how can this occur if they are in different virtual addrspaces?

- we make the kernel's vm overlap with process' virtual memories.

OS/161 MEMORY



kuseg: user virtual memory

kseg0 (512MB): kernel data structs, stacks

kseg1 (512MB): addressing devices

kseg2 (1GB): not used

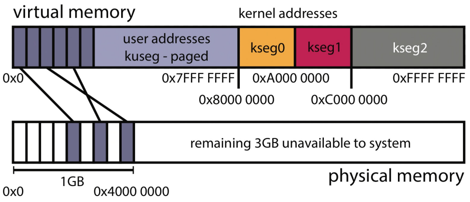
these are divided into frames, managed by `coremap`

kuseg

1 kuseg is paged.

2 The kernel maintains page → frame mappings for each process.

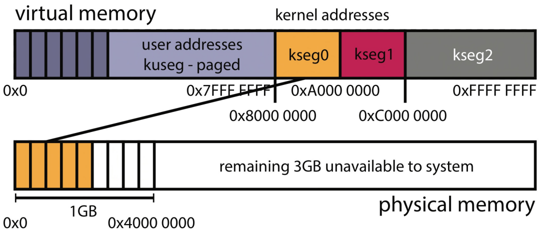
3 TLB is used to translate `kuseg` vaddr to physical ones.



kseg0

1 To translate kseg0 addresses to physical ones, just subtract 0x80000000 from the vaddr.

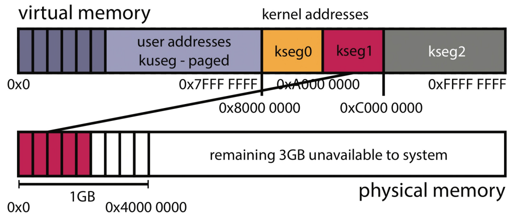
2 kseg0 maps to the first 512MB of physical memory, but it might not use all this space.



kseg1

1 To translate kseg1 addresses to physical ones, subtract 0xA0000000 from the virtual address.

2 kseg1 also maps to the first 512MB of physical memory, though it might not use all this space.



EXPLOITING SECONDARY STORAGE

1 Goal:

① Allow virtual addrspaces that are larger than the physical addrspace;

② Allow greater processor utilization by using less of the available primary memory for each process.

2 Method:

① Allow pages from VMs to be stored in secondary storage (HDD/SSD); &

② Swap pages/segments between secondary storage & primary memory so they are in the latter when needed.

RESIDENT SETS

The set of virtual pages present in physical memory is the "resident set" of a process.

PRESENT BITS

1 To track which pages are in physical mem, we assign each PTE a "present bit".

- valid=1, present=1 ⇒ page is valid and in memory
- valid=1, present=0 ⇒ page is valid, but not in memory
- valid=0 ⇒ invalid page

PAGE FAULTS (PF)

1 If an accessed page's present bit = 0, then the page is not in primary memory. This is called a "page fault".

2 With a hardware-managed TLB:

- the MMU detects this situation when PTE is checked
- it generates an exception, handled by the kernel.

3 With a software-managed TLB:

- kernel detects this when it checks the PTE after a TLB miss.

(TLB should not contain any entries not present in RAM.)

4 When a PF occurs, the kernel must:

① Swap the page into memory from secondary storage, possibly evicting another page;

② Update the `present` bit in the page's PTE; &

③ Return from the exception so the application can retry the VM access (that caused the PF).

REDUCE PFs

1 The PF frequency impacts the average memory access time.

2 Idea: To improve performance of VM with on-demand paging, reduce the occurrence of PFs.

3 Strategies:

① Limit # of processes;

- so there is enough memory per process

② Use a "replacement policy";

- be smart about which pages are kept in physical memory & which are evicted

③ Hide latencies by prefetching pages before a process needs them.

FIFO

Idea: Replace the page that has been in memory the longest.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 0	a	a	a	d	d	d	e	e	e	e	e	e
Frame 1		b	b	b	a	a	a	a	a	c	c	c
Frame 2			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

which page was referenced

was there a page fault?

a was the "first in", so we evict a & swap it with d ("first out").

b is now the "first in", so we evict b & swap it with a.

MIN

1 Idea: replace the page that will not be referenced for the longest time.

2 This is optimal for on-demand paging.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 0	a	a	a	a	a	a	a	a	a	c	c	c
Frame 1		b	b	b	b	b	b	b	b	b	d	d
Frame 2			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

c won't be referenced for the longest time, so replace it with d.

a won't be referenced again, so replace it with c.

3 This requires knowledge of the future.

LOCALITY

1 "Temporal locality" — programs are more likely to access pages that they have recently accessed.

2 "Spatial locality" — programs are more likely to access parts of memory close to parts of memory they have accessed recently.

3 Locality keeps PF rates low.

LEAST RECENTLY USED (LRU)

1 Idea: evict the page not used in the longest period of time.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 0	a	a	a	d	d	d	e	e	e	c	c	c
Frame 1		b	b	b	a	a	a	a	a	a	d	d
Frame 2			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

e was least recently used, so replace it with c.

2 Challenges:

① It must track usage & find a max value, which is expensive.

② The kernel is unaware which pages a program is using unless there is an exception.

CLOCK REPLACEMENT

Solution: have the mmu track page accesses in hardware.

- Add a `use`/`reference` bit to each PTE.

The use bit :

① is set by mmu each time the page is used

② can be read/cleared by the kernel.

Idea: Use a "victim pointer" that cycles through the page frames, & it moves whenever a replacement is necessary.

```
while (use bit of victim is set) {
  clear use bit of victim            // get a 2nd chance
  victim = (victim + 1) % num_frames  // go to next frame
}
evict victim                          // its use bit is 0
victim = (victim + 1) % num_frames
```

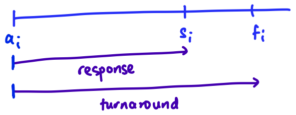
Example:

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 0	a	a	a	d	d	d	e	e	e	c	c	c
Frame 1		b	b	b	a	a	a	a	a	a	d	d
Frame 2			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x
victim				0	1	2	0			0	1	2

Topic 6: Scheduling

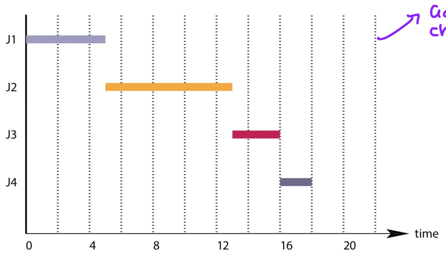
JOB SCHEDULING PROBLEM

1. A set of jobs $J_1, \dots, J_n$ needs to be executed using a single server.
   - Only one job can run at a time;
   - The server can switch from one job to another instantly & at any time.
2. Each job J_i has
   ① an "arrival time" a_i — when J_i becomes available
   ② a "run time" r_i — how long J_i needs to finish
3. Output:
   ① A job "scheduler" decides which job should be running on the server at each point in time.
   ② These decisions determine a "start time" (s_i) & "finish time" (f_i) for each job J_i.
4. Performance metrics:
   ① "Response time" of the scheduler, $s_i - a_i$;
   ② "Turnaround time" of the scheduler, $f_i - a_i$.



FIRST COME, FIRST SERVE (FCFS)

Idea: Run jobs in arrival time order.

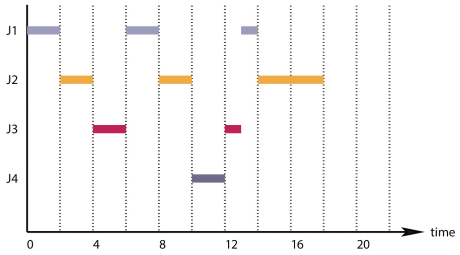


Gantt chart.

Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

ROUND-ROBIN (RR)

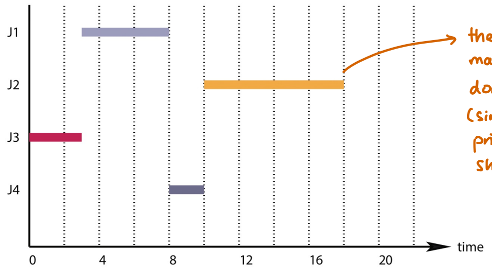
Idea: FCFS with preemption.
(This is used by OS/161's scheduler.)



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SHORTEST JOB FIRST (SJF)

1. Idea: Run the shortest job first.
2. This minimizes average turnaround time but "starvation" is possible.

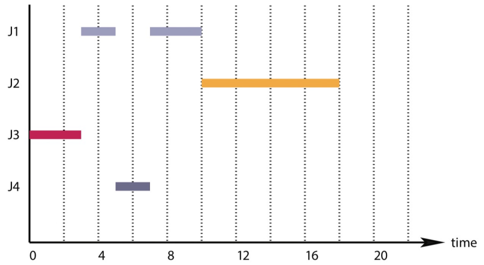


the longest job may never get done ! (since we prioritize shorter jobs)

Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SHORTEST REMAINING TIME FIRST (SRTF)

1. Idea: Use SJF but with pre-emption.



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

2. Note starvation is still possible.

PERFORMANCE METRICS

Turnaround Time (i.e. finishing time - arrival time) for jobs J1–J4

- *FCFS* (5-0) + (13-0) + (16-0) + (18-5) = 47. Average = 47/4 = 11.75
- *SJF* (3-0) + (8-0) + (18-0) + (10-5) = 34. Average = 34/4 = 8.5
- *RR* (14-0) + (18-0) + (13-0) + (12-5) = 52. Average = 52/4 = 13.0
- *SRTF* (10-0) + (18-0) + (3-0) + (7-5) = 33. Average = 33/4 = 8.25

Response Time (i.e. start time - arrival time) for jobs J1–J4

- *FCFS* (0-0) + (5-0) + (13-0) + (16-5) = 29. Average = 29/4 = 7.25
- *SJF* (3-0) + (10-0) + (0-0) + (8-5) = 16. Average = 16/4 = 4.0
- *RR* (0-0) + (2-0) + (4-0) + (10-5) = 11. Average = 11/4 = 2.75
- *SRTF* (3-0) + (10-0) + (0-0) + (5-5) = 13. Average = 13/4 = 3.25

CPU SCHEDULING

1 Here, the jobs to be scheduled are the threads.

2 Differences from normal scheduling:

① run-times are unknown

② threads are sometimes unrunnable (they might be blocked)

③ they may have different priorities

3 Objectives of the scheduler: balance

① "responsiveness": ensure threads run regularly

② "fairness": sharing of the CPU

③ "efficiency": account that there is a cost in switching

4 How to handle priorities?

① Schedule the highest priority thread; or

② "Weighted fair sharing".

- if p_i = priority of i^{th} thread
- try to give each thread a share of the CPU in proportion to its priority ($\frac{p_i}{\Sigma p_j}$)

MULTI-LEVEL FEEDBACK QUEUES (MLFQ)

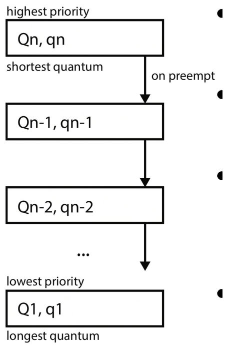
1 Objective: Good responsiveness for interactive threads, -eg I/O threads whilst also making good progress with non-interactive threads.

2 How do we determine if a thread is interactive?

3 Idea: Interactive threads are frequently blocked (waiting for user input etc)

4 So, give higher priority to threads that block frequently, so they run when they are ready.

5 Algorithm:

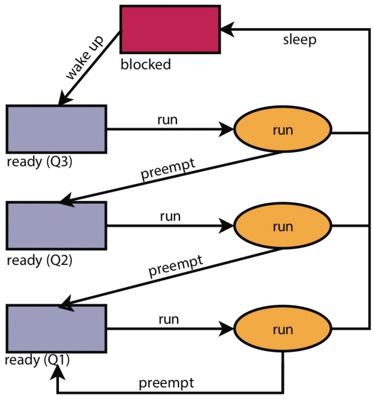


- we have n round-robin queues Q_i in decreasing order of priority.
- threads in Q_i use quantum q_i, & $q_1 > q_2 > \cdots > q_n$
- The scheduler selects threads from the highest priority queue to run. (Q_{n-1} threads run if Q_n is empty).
- Preempted threads are put at the back of the next lower-priority queue
- When a thread wakes after blocking, put it in the highest priority queue.

6 Since interactive threads tend to block frequently, they tend to stay in the higher priority queues.

7 Non-interactive threads sift toward the bottom.

8 Example: (3 queues)



*each level has its own "run" & "ready" queue, but they all share the "blocked" queue.

Note: To prevent starvation, all threads are periodically placed in the highest priority queue.

COMPLETELY FAIR SCHEDULER (CFS)

1 Idea: Assign each thread a weight, & the scheduler ensures each thread gets a share of the processor proportional to its weight.

2 If

- a_i = actual amount of time the scheduler has allowed T_i to run for
- w_i = weight of T_i

We ideally want $\frac{a_i}{w_i} = \frac{a_j}{w_j} \ \forall T_i, T_j$.

3 Then the thread with the smallest $\frac{a_i}{w_i}$ ratio should run next to increase its share of the processor time.

4 To do this, track the "virtual run-time" of each runnable thread.

$$\boxed{\text{virtual run-time of } T_j = a_i \frac{\Sigma w_j}{w_i}.}$$

So we want to run the thread with the lowest virtual run-time.

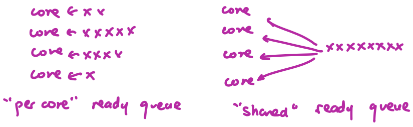
5 When a thread becomes runnable, initialize its virtual run-time to some value between the min & max virtual run-times of the threads always runnable.

MLFQ VS CFS

1 In MLFQ, the quantum depends on the thread's priority.

2 In CFS, the quantum is always the same for all threads & priorities.

SCHEDULING ON MULTI-CORE PROCESSORS



"Per core" ready queue scales better as the # of cores increases.

- access to a shared ready queue is a critical section
- so mutex is required

CPU CACHE AFFINITY

1 As a thread runs, data is loaded into CPU cache(s).

2 Usually, a core will have some memory caches of its own, & some it shares with other cores.

3 Moving the thread to another core means data must be reloaded into that core's caches.

4 As the thread runs, it acquires an affinity for one core because of the cached data.

Topic 7: I/O

DEVICES

- "Devices" are how a computer receives input from the outside world & produces output for the outside world.

BUS

- A "bus" is a communication pathway between various devices in a computer.
  - ① "internal bus": for communication between the CPU & RAM.
    - it is fast & close to the CPU
  - ② "peripheral bus": allows devices in the computer to communicate.

BRIDGE

- A "bridge" connects 2 different buses.

DEVICE REGISTERS

1. Communication with devices are carried out through "device registers".
2. Three primary types:
   - ① "status": tells you something about the device's current state.
   - ② "command": issue a command to the device by writing a particular value to this register.
   - ③ "data": used to transfer larger blocks of data to/from the device.
3. These might be combined:
   - ① a "status & command" register is read to discover the device's state & written to issue the device a command.
   - ② a "data buffer" — sometimes combined or separated into data in/out buffers.

SYS/161 TIMER/CLOCK

- The clock is used in pre-emptive scheduling.

Offset	Size	Type	Description
0	4	status	current time (seconds)
4	4	status	current time (nanoseconds)
8	4	command	restart-on-expiry
12	4	status and command	interrupt (reading clears)
16	4	status and command	countdown time (microseconds)
20	4	command	speaker (causes beeps)

these are in bytes

* the location is relative to a base value.
eg base = 0x1FE00000.

SERIAL CONSOLE

- The character buffer is used to write outgoing characters & read incoming characters.

Offset	Size	Type	Description
0	4	command and data	character buffer
4	4	status	writeIRQ
8	4	status	readIRQ

IRQ = interrupt request

* if a write occurs when a write is already in progress, device exhibits UB

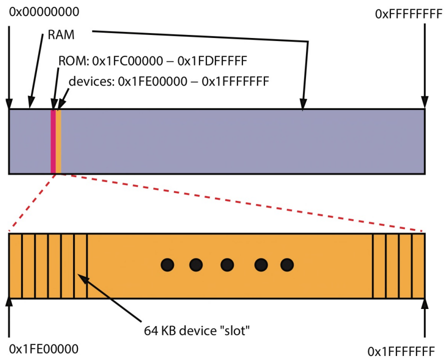
DEVICE DRIVER

1. A "device driver" is a port of the kernel that interacts with a device.
2. Communication happens by reading from or writing to the command, data & status registers.
3. Methods of interacting with devices:
   - ① "Polling": the kernel driver repeatedly checks the device status
   - ② Interrupts.
     - kernel does not wait for device to complete command
     - request completion is taken care of interrupt handler
     - ie device updates a status register to indicate if it was successful, then generates an interrupt.

ACCESSING DEVICES

1. How can a device driver access the device registers?
2. Port-mapped I/O:
   - uses special assembly I/O instructions
   - device registers are assigned port #s, which correspond to regions of memory in a smaller separate addrspace.
3. Memory-mapped I/O:
   - each device registe has a physical memory address
   - device drivers can read/write to these device registes using load/store instructions
4. A device may use both I/O options.

OS/161 PHYSICAL ADDRSPACE



LARGE DATA TRANSFER TO/FROM DEVICES

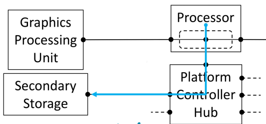
1 "Program-controlled I/O" (PIO)

- device driver moves data between memory & a buffer on the device
- CPU is used to transfer data

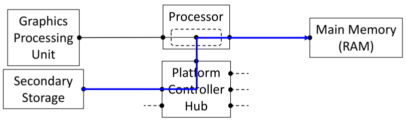
2 Direct memory access (DMA)

- device itself is responsible for moving data to/from memory
- CPU is not used to transfer data, & can do something else
- this is used for block data transfers between devices.

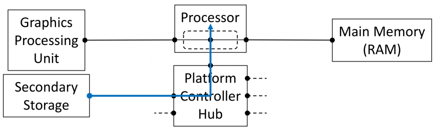
3 Example:



- processor initiates data transfer, & gives starting addr & amount of data to transfer



- controller directs data transfer



- controller interrupts processor when transfer is complete.

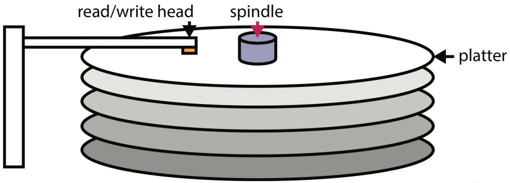
PERSISTANT STORAGE

"Persistant / non-volatile storage" is any device where data persists even when the device is without power.

- primary memory (RAM) is not persistant
- secondary memory (hard disk) is persistant

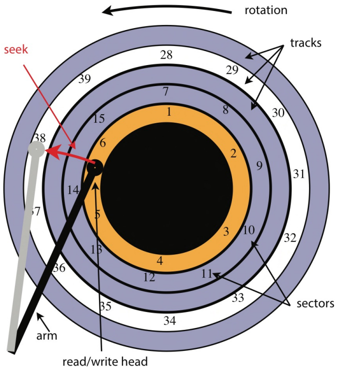
HARD DISKS

1 HDDs are persistant storage devices.



- to read/write, the platter & R/W head must move
- this takes milliseconds

2 Logical view:



- a disk is an array of numbered blocks of identical size (~512B)
- blocks are the unit of transfer between disk & memory
- one or more contiguous blocks can be transfered in a single operation.

3 Physical view:

- A HDD consists of 1-4 platters/disks.
- Each platter has two surfaces.
- The surface is broken up into a series of concentric circles.
  - "tracks" if one surface
  - "cylinders" if same radius on each surface
- Each track is broken up into a series of arcs called "sectors"
- Each surface also has a "disk head" / "read/write head" to read/write data from it.
- All disk heads are put in position by a single disk/actuator arm.

COST MODEL FOR DISK I/O

💡1 The time it takes to move data to/from a disk involves

① the "seek time": the time it takes to move the read/write heads to the appropriate track

- depends on the seek distance: distance in tracks between previous & current request
- ranges from 0 to cost of max seek dist

② the "rotational latency": the time it takes for the desired sectors spin to the read/write heads

- depends on rotational speed of disk
- ranges from 0ms to cost of single rotation

③ the "transfer time": the time it takes until desired sectors spin past read/write heads

- depends on rotational speed of disk & amount of data accessed

💡2 $\boxed{\text{Request service time = seek time + rotational latency + transfer time}}$

REQUEST SERVICE TIME EXAMPLE

Parameters

- Disk Capacity: 2^{32} bytes
- Number of Tracks: 2^{20} tracks
- Number of Sectors per Track 2^{8} sectors
- Rotations per Minute (RPM): 10000 RPM
- Maximum Seek (time): 20 milliseconds

💡1 How many bytes in track?

$$\# = \frac{\text{disk capacity}}{\text{\# tracks}} = \frac{2^{32}}{2^{20}} = 2^{12} \text{ B/track}$$

💡2 How many bytes in a sector?

$$\# = \frac{\text{bytes/track}}{\text{sectors/track}} = \frac{2^{12}}{2^{8}} = 2^{4} \text{ B/sector}$$

💡3 Max rotational latency?

$$\text{max} = \frac{60}{\text{RPM}} = \frac{60\text{s}}{10000} = 6.006\text{s}$$

💡4 Average seek time?

$$\text{avg} = \frac{\text{max seek}}{2} = \frac{20\text{ms}}{2} = 10 \text{ ms}$$

💡5 Average rotational latency?

$$\text{avg} = \frac{\text{max latency}}{2} = \frac{6\text{ms}}{2} = 3\text{ms}$$

💡6 Cost to transfer 1 sector?

$$\text{sector latency} = \frac{\text{max latency}}{\text{\# sectors on track}} = \frac{6\text{ ms}}{2^8} = 0.0195 \text{ ms/sector}$$

💡7 Expected cost to read 10 consecutive sectors?

$$\begin{aligned}\text{cost} &= \text{average seek + average rotational latency + transfer time} \\ &= 10 + 3 + 10(0.0195) \\ &= 13.195 \text{ ms.}\end{aligned}$$

PERFORMANCE IMPLICATIONS OF DISK CHARACTERISTICS

💡1 Note:

① Larger transfers are more efficient than smaller ones.

- as time it gets to get to the region being read is amortized over many blocks of data.

② Sequential I/O is better than non-sequential I/O.

- sequential I/O operations eliminate need for most seeks.

💡2 While sequential I/O is not always possible, we can group requests to try to reduce average request time.

DISK HEAD SCHEDULING

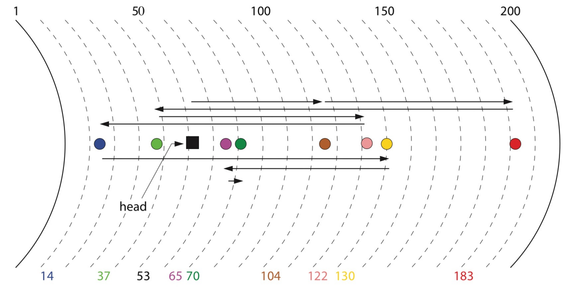
💡1 Goal: Reduce seek times by controlling order in which requests are serviced.

- we need queue of outstanding disk requests.

FIRST-COME-FIRST-SERVED (FCFS)

💡1 **Idea:** Service requests in the order they arrive.

💡2 **Tradeoffs:** Fair & simple, but no optimization for seek time.

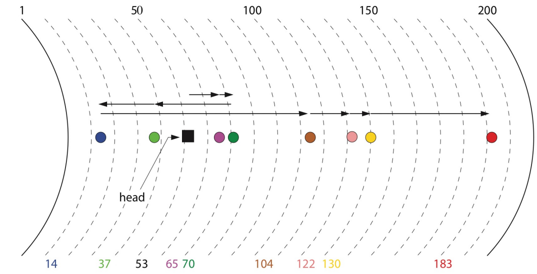


The service order is: 104 183 37 122 14 130 65 70

SHORTEST SEEK TIME FIRST (SSTF)

💡1 **Idea:** Service closest request first ("greedy").

💡2 Tradeoffs: reduce seek times, but some requests may starve.

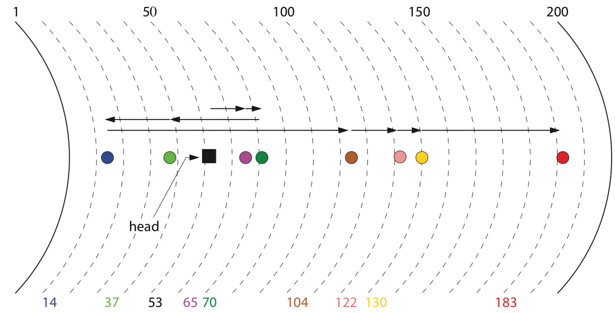


The service order is: 65 70 37 14 104 122 130 183

ELEVATOR ALGORITHMS (SCAN)

💡1 **Idea:** Disk head moves in 1 direction until there are no more requests in front of it, then reverses direction.

💡2 Tradeoffs: reduces seek times & avoids starvation.



The service order is: 65 70 37 14 104 122 130 183

SYS/161 DISK CONTROLLER

Offset	Size	Type	Description
0	4	status	number of sectors
4	4	status and command	status
8	4	command	sector number
12	4	status	rotational speed (RPM)
32768	512	data	transfer buffer

reports present state of disk

1 To perform an operation:

① Store sector # in `sector #` register

② Then write `read-in-progress` or `write-in-progress` into `status`.

2 When the disk controller is reporting a completed operation, an interrupt is generated.

READ FROM/TO SYS/161 DISK

1 Constraints:

① OS/161 allows only one thread at a time to access the disk.

② The thread that initiates "write" should wait until that write is completed before continuing (similarly for "read").

2 We use 2 semaphores:

① `disk` : used so only one thread at a time can use the disk.

② `disk_completion` : used so the thread sleeps on after it has acquired exclusive access to the disk, but is waiting for its request to be completed.

* once the disk has completed the request, device driver will call V(disk-completion).

3 Device driver write handler (called by thread wanting to write):

```
// allow only one disk request at a time
   P(disk)
   copy data from RAM to device transfer buffer
   write target sector number to sector number register
   write write command to the status register
// wait for request to complete
   P(disk_ completion)
// when request completed, allow others access to disk
   V(disk)
```

4 Interrupt handler for disk device (called by kernel):

```
// make the device ready again
   write disk status register to acknowledge completion
   V(disk_completion)
```

5 Device driver read handler (called by thread wanting to read):

```
// allow only one disk request at a time
   P(disk)
   write target sector number to sector number register
   write read command to the status register
// wait for request to complete
   P(disk_completion)
   copy data from device transfer buffer to RAM
// when request completed, allow others access to disk
   V(disk)
```

6 Interrupt handler for disk device (called by kernel):

```
// make the device ready again
   write disk status register to ack completion
   V(disk_completion)
```

SSD & FLASH MEMORY

1 Idea: Use integrated circuits to persistant storage instead of magnetic surfaces.

↳ no mechanical parts.

2 "Flash memory" uses quantum properties of electrons.

↳ compared to DRAM — requires constant power to keep electrons.

3 Flash memory is divided into blocks & pages:

- 2,4,8 KB pages
- 32KB-4MB blocks

4 For flash memory reads/writes at page level:

① Pages are initialized to 1s:

② Transition 1→0 can occur at the page level.

5 But overwriting/deleting must be done at the block level.

① A high voltage is required to switch 0→1.

② Can apply high voltage at the block level.

WRITING & DELETING FROM FLASH MEMORY

1 Naive solution (slow):

- read whole block into RAM
- reinitialize block
- update block in RAM & write back to SSD

2 SSD controller handles requests (faster):

- page to be deleted/overwritten is marked as invalid
- write to an unused page
- update translation table
- requires garbage collection

3 SSD limitations:

① Each block of an SSD has a limited # of write cycles before it becomes read-only.

② SSD controllers perform wear levelling, distributing writes evenly across blocks

- so blocks wear down at an even rate.

③ Defragmenting can be harmful to a SSD's lifespan.

PERSISTENT RAM

1 In "persistent RAMs", data is preserved in the absence of power.

2 These can be used to improve the performance of secondary storage.

- traditional CPU caches are small; not effective for caching many disk blocks' worth of data
- volatile RAM can cache file system info reserved for process addrspaces.
- we cache file system info in persistent RAM.

Topic 8: File Systems

FILES

1. Files are persistent, named data objects.
2. Data consists of a sequence of numbered bytes. (No other assumptions can be made.)
3. Files have associated meta-data.
   eg owner, file type, etc

FILE SYSTEMS

1. "File systems" are the data structures & algorithms used to store, retrieve & access files.
2. A file system can be separated into 3 layers:
   ① "Logical file system": high-level API, what a programmer sees
   eg fopen, fscanf, etc
   - this manages file system information;
   eg which files are open for writing & reading
   ② "Virtual file system": abstraction of lower level file systems
   - presents multiple different underlying file systems (eg HDD, SDD, etc) to the user as one
   ③ "Physical file system": how files are actually stored on physical media
   - eg track, sector, etc

BASIC FILE INTERFACE

1. Common file operations:
   ① 'open' — returns a file descriptor
   - other file operations for this process requires this descriptor as a parameter
   - to identify the file.
   ② 'close' — invalidates a valid file descriptor for a process.
   - kernel tracks which file descriptors are currently valid for each process.
   ③ 'read' — copies data from a file into a virtual address space.
   ④ 'write' — copies data from a virtual address space to a file.
   ⑤ 'seek' — enables non-sequential reading/writing.
   ⑥ Get/set file meta-data.
   - eg (Linux) fstat, chmod, ls -la

FILE POSITION & SEEKS

1. Each open file (ie valid descriptor) has an associated file position.
   - this starts at byte 0 when file is opened.
2. Read & write operations
   - start from the current file position; &
   - updates the current file position as bytes are read/written.
   ↳ this makes sequential file I/O easy.
4. 'seek' changes the file position of a descriptor to allow non-sequential I/O.
   - note seek does not throw an exception if the position is invalid
   - but the subsequent read/write operation will.

SEQUENTIAL FILE READING EXAMPLE

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f, (void *)buf, 512);
}
close(f);
```

this shows the file is open for only reading.

read first 512 x 100 bytes of a file, 512 bytes at a time

FILE READING USING SEEK EXAMPLE

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f, (100-i)*512, SEEK_SET);
  read(f, (void *)buf, 512);
}
close(f);
```

read first 51200 bytes, 512 bytes of a time, in reverse order.

DIRECTORIES & i-NUMBERS

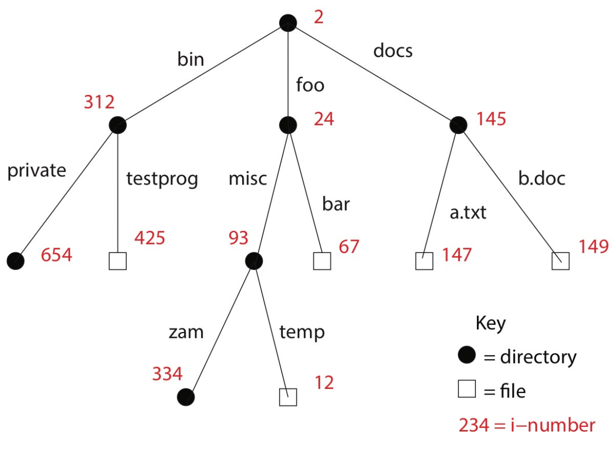
1. A "directory" maps file names (strings) to i-numbers.
2. An "i-number" (index number) is a unique identifier for a file/directory (in a file system).
   - given an i-number, the file system can find the data & metadata for the corresponding file.
3. Directories provide a way for applications to group related files.

DIRECTORIES AS TREES

1. A file system's directories can be viewed as a tree, with a single root directory.
   - files are always leaves
   - directories can be interior nodes or leaves.
2. Only the kernel can directly edit directories.
   - kernel should not trust the user with direct access to data structures the kernel relies on.

PATHNAMES

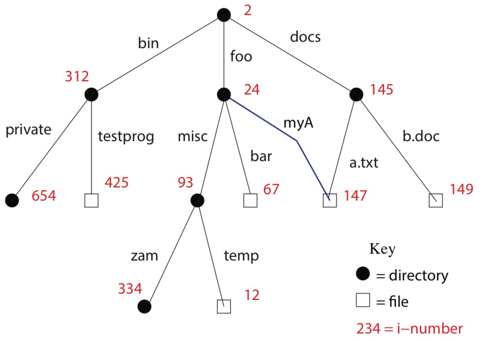
1. Files may be identified via "pathnames", that describe a path through the directory tree from the root directory to the file.
   - directories also have pathnames.



/docs/b.doc is the path for the file with i-number 149.

HARD LINKS

1. A "hard link" is an association between a name (string) & i-number.
2. When a file is created, a hard link to that file is also created.
   - eg open("/docs/a.txt", "O_CREAT | O_TRUNC")
   - this opens the file & creates a hard link to that file in /docs
3. When a file is created, additional hard links can be made to it.
   eg link /docs/a.txt /foo/myA
   - this creates a new hard link myA in directory foo
   - to the i-number of the file /docs/a.txt.
4. Each file has a unique i-number, but may have multiple pathnames.



/foo/myA and /docs/a.txt are two different paths to file 147

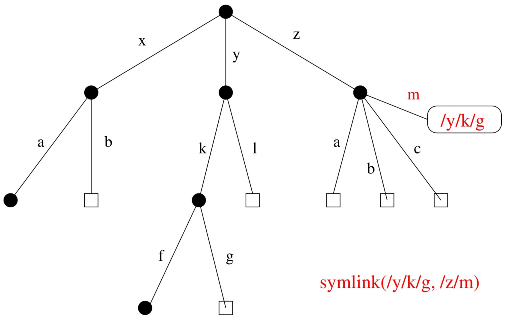
5. To avoid cycles, we disallow linking to a directory.

UNLINKING

1. Hard links can be removed.
   eg unlink /docs/b.doc
2. The FS ensures that hard links have "referential integrity":
   if the link exists, then the file it refers to also exists.
   - the kernel keeps a count of how many hard links there are to an existing file
   - a file is deleted when its last hard link is removed (ie the count becomes 0).

SYMBOLIC / SOFT LINKS

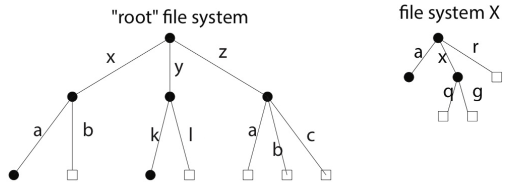
1. A "soft link" is an association between a name (string) and pathname.
   - we can do this via symlink(/z/a, /y/k/m)
2. If an application wants to open /y/k/m, the FS will
   - recognize it as a symbolic link; and
   - attempt to open /z/a.
3. Note referential integrity is not preserved;
   ie /z/a may not exist!



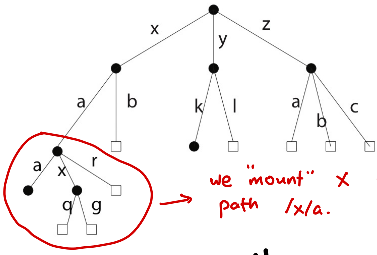
symlink(/y/k/g, /z/m)

MULTIPLE FILE SYSTEMS

1. A system might have multiple file systems.
   - eg USB flash drives
2. We need a global file namespace; ie
   - uniform across all the file system
   - independent of physical location
3. Methods:
   ① Use file-system name in the file name
   eg C:/user/ ...
   ② Create a single hierarchical namespace that combines the namespaces of multiple file systems
   ie using `mount`.
4. "Mounting" does not take two file systems into one; it just combines the namespaces into one hierarchical one.



result of mount (file system X, /x/a)



we "mount" X to the path /x/a.

FILE SYSTEM IMPLEMENTATION

1. What is stored persistently (ie in secondary storage)?
   - file data & meta-data
   - directories & links
   - file system meta-data
2. What is non-persistent?
   - per process open file descriptor table:
     - file descriptor
     - file position for each open file
   - system wide:
     - open file table (info about files currently open)
     - cached copies of persistent data

FILE SYSTEM EXAMPLE:
VERY SIMPLE FILE SYSTEM (VSFS)

1 Goal: Create a small file system to show what components are needed & how they are used.

① Use an extremely small disk.

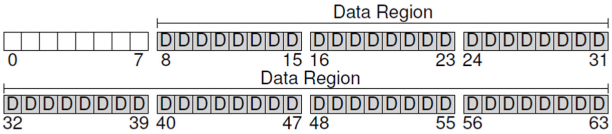
- this has a sector size of 512 bytes.
- note: disks are usually sector addressable, ie can read/write to any sector
- disk will be 256 kB → so we have 512 total sectors.

② Group every 8 consecutive sectors into a block (4kB)

- better spatial locality (fewer seeks)
- reduces # of block pointers
- a 4 kB block is a convenient size for on-demand paging
- 64 blocks on the disk

2 We want to decide ahead of time how much space to reserve for data vs meta-data.

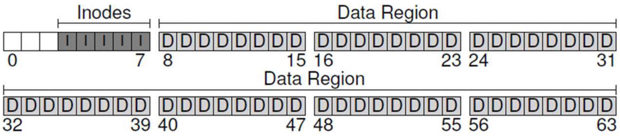
- most of the blocks should be for storing a file's contents.



* we reserve the first 8 blocks for meta-data & the last 56 blocks for file data.

3 We also need some way to map files to their data blocks.

- create an array of i-nodes, where each i-node contains a file's metadata.
- index into the i-node array is the file's index number.



* we use 256 bytes for each i-node & dedicate 5 blocks for i-nodes.

↳ this allows for 80 total i-nodes $\left(\frac{2^{12}}{2^{8}} \times 5\right)$

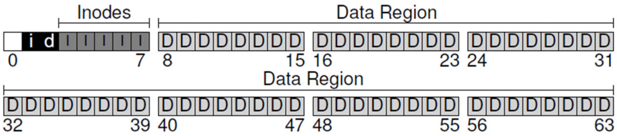
↳ so ≤ 80 files

4 We also need to track which i-nodes & data blocks are in use.

- we use a bitmap for each:
  - ① i: a block containing the bitmap tracking i-nodes in use
  - ② d: a block containing the bitmap tracking data blocks in use

  (we could use a free list instead of a bitmap).
- a block size of 4KB can track 4K × 8 = 32K i-nodes or data blocks.

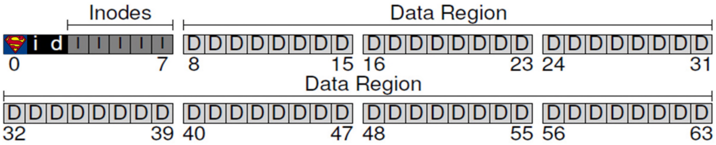
  ↳ but we only need to track ≤ 80 of them.



5 Reserve the first block as the "superblock".

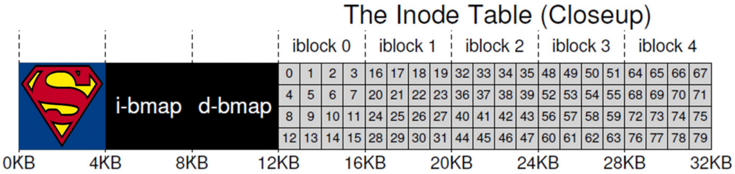
- contains meta-info about the entire file system

  eg # of i-nodes/blocks in the system, location of i-node bitmap/data block bitmap, etc



6 Close-up of first 8 blocks of FS:

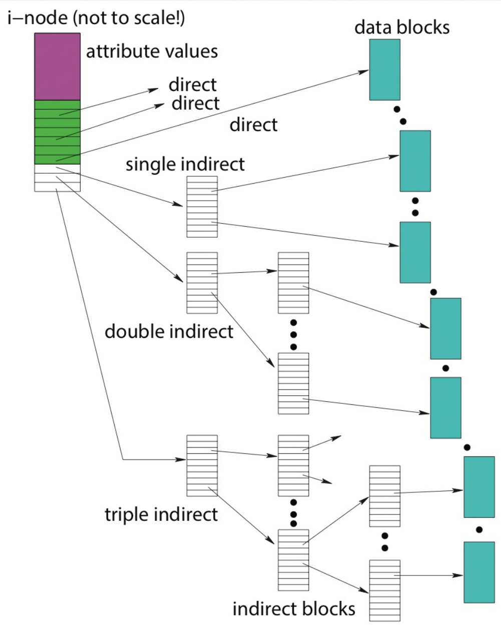
The Inode Table (Closeup)



0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79

i-NODES

1. An "i-node" is a fixed size index structure that holds the file meta-data & pointers to the data blocks.
   - fields of i-nodes include file type, permissions, length; time of last file access; # of hard links to file; & ptrs to data blocks.
2. For small files, pointers in the i-node are sufficient to point to all data blocks; ie "direct data block pointers".
3. For larger files, we use "indirect data block pointers".



4. Idea: Use both direct & indirect block pointers.
   ① If disk blocks are referenced with 4B address, then
   - there are 2^{32} blocks each of 4KB
   - so max disk size is $2^{32} \times 2^{12} = 2^{44}$ = 16 TB.

   ② In VSFS, an i-node is 256B.
   - reserve room for 12 ptrs to blocks; "direct pointers".
   - each pointer points to a different block for storing user data.
   - pointers are ordered; i^{th} pointer to the i^{th} block in the file.

   ③ If we only have direct pointers:

   $$\boxed{\text{max file size} = 12 \times 4KB = 48\,KB.}$$

   ④ We then introduce an indirect pointer, which points to a block full of direct pointers.

   ⑤ A 4KB block of direct pointers hold 1024 4B pointers. So with 12 direct & 1 indirect pointers:

   $$\boxed{\text{max file size} = (12+1024) \times 4KB = 4144\,KB}$$

   ⑥ If we need more space, we can add a double indirect pointer, which points to a block full of indirect pointers.

   ⑦ With 12 direct, 1 indirect & 1 double indirect pointers:

   $$\boxed{\text{max file size} = (12+1024+1024^2) \times 4KB \cong 4GB.}$$

   ⑧ If we need more space, we can use a "triple indirect pointer".

DIRECTORIES

1. Directories are implemented as a special type of file that contains directory entries, pairing up
   ① An i-number; &
   ② A file name.
2. Directory files can be read by application programs.
3. Directory files are only updated by the kernel, in response to file system operations.
   eg create file/link.

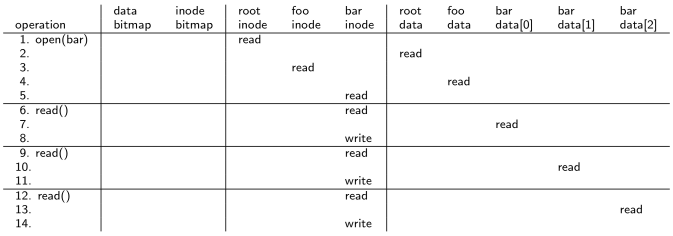
IN-MEMORY NON-PERSISTENT STRUCTURES

1. To speed up file access, the kernel keeps some information in RAM which is updated as processes access files.
2. Per process:
   ① "Descriptor Table"
   - tracks which file descriptors does the process have open
   - which file does each open descriptor refer?
   - current file position for each descriptor?
3. System wide:
   ① "Open file table": files currently open by any process
   ② "i-Node cache": in-memory copies of recently-used i-nodes
   ③ "Block cache": in-memory copies of data blocks & indirect blocks.

READING FROM A FILE

eg reading /foo/bar

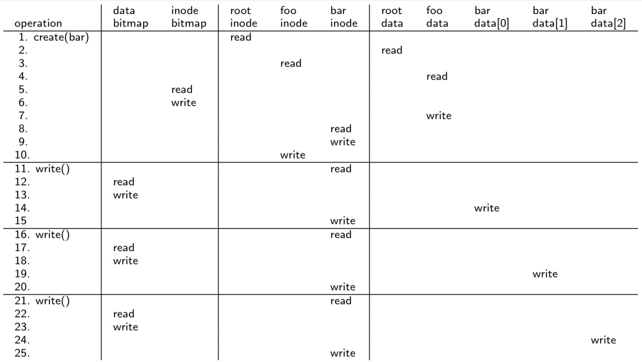
1. First, the root i-node is read.
   - this provides the location of root's data block
   - which stores the root directory
2. The root's data block is read to find the i-number for ‘foo’.
   - a directory associates a string (file / directory name) to an i-number
   - we assume directory fits in a single block.
3. Then, read ‘foo’s i-node, which provides the location of foo's data.
4. Read ‘foo’s data (a directory) to find bar's i-number.
5. Similarly, read ‘bar’s i-node.
   - check the file permissions
   - a file descriptor is returned & added to the process's file descriptor table
   - file is added to the kernel's open file table.
6. To read data from /foo/bar: ‘bar’s i-node is read and a pointer to ‘bar’s data block is found.
7. A data block for /foo/bar is read. (The 0^{th} data block).
8. ‘bar’s i-node is written to update the access time.
9. We repeat the previous three steps to read subsequent data blocks.



operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read							
2.						read				
3.				read						
4.							read			
5.					read					
6. read()					read					
7.								read		
8.					write					
9. read()					read					
10.									read	
11.					write					
12. read()					read					
13.										read
14.					write					

CREATING A FILE

eg creating the file `bar` in directory `foo`



operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. create(bar)			read							
2.						read				
3.				read						
4.							read			
5.		read								
6.		write								
7.							write			
8.					read					
9.					write					
10.				write						
11. write()					read					
12.	read									
13.	write									
14.								write		
15					write					
16. write()					read					
17.	read									
18.	write									
19.									write	
20.					write					
21. write()					read					
22.	read									
23.	write									
24.										write
25.					write					

note: when writing a partial block that block must be read first; but when writing an entire block, no read is required.

CHAINING

VSFS uses a per-file index (direct & indirect pointers) to access blocks.

Alternative approaches:

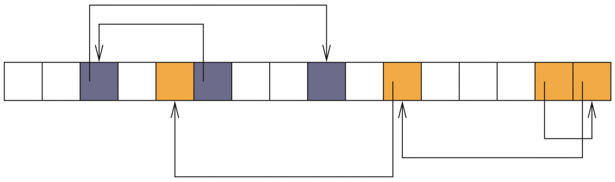
① Chaining

→ each block includes a pointer to the next block

② External chaining

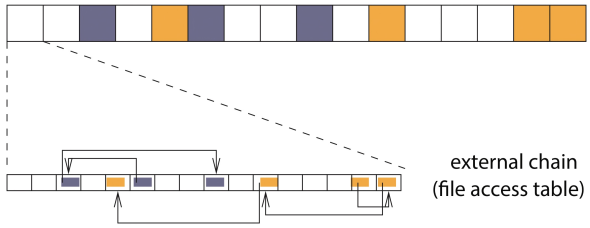
→ the chain is kept as an external structure.

For chaining, the directory table contains the name of the file paired with its starting block, and possibly its end block.



- this is acceptable for sequential lookup, but very slow for random access
- because we need to go through the file sequentially to get to a particular location.

To improve performance, in external chaining, introduce a "file access table" (FAT) that specifies all the chains in one big table.



- when doing random access, this approach reduces the # of blocks read
- by keeping this table in its own block rather than having one pointer / block.

FILE SYSTEM DESIGN

Idea: Design our FS to be efficient for the "common case".

- we typically only deal with small files.

But we should also be able to handle

- a large # of small files; or
- a small # of large files.

PROBLEMS CAUSED BY FAILURES

A single logical FS operation may require several disk I/O operations.

eg deleting a file ⇒ remove directory entry, remove file index, etc.

But if some but not all of these changes are reflected on the disk after system failure, it could destroy in-memory file system structures.

So persistent structures should be "crash consistent"; ie they should be consistent when a system restarts after a failure.

Methods:

① Special-purpose consistency checkers

- runs after a crash but before normal operations resume
- finds & attempts to repair inconsistent file system data structures

eg file with no directory entry, free space not marked as free

② Journaling

- uses "write-ahead logging":
- first record the file meta-data changes in a journal
- so sequences of changes can be written to disk in a single operation
- after the changes have been journaled, update the disk data structures.
- after a failure, redo the journaled updates in case they were not completed before the failure.