Operating Systems

Running Syllabus

Last updated: 2018-03-27

Status: IN PROGRESS

Calendar Table

Document status

UP-TO-DATE - Document active and updated
IN PROGRESS - Document active and in the process of being updated
INACTIVE - Document inactive

Legend

Cx	C programming language assignment
Px	Pintos programming assignment
OSPP	Textbook “Operating Systems: Principles and Practice”
Pintos	Pintos Manual

Dates

If only one date/time, it holds for both UCD/MSUD. Else:

For assignments

UCD date/time MSUD date/time

Dates for workshops

UCD date 1, UCD date 2 MSUD date

Wk	What	Date | Date/time open	Date/time due
1	C review		
2	C review, OS Intro (OSPP 1)		
2	C1	Mon, 2017-01-23 00:00	Tue, 2017-02-07 23:59 Thu, 2017-02-09 23:59
3	OS Intro, threads (OSPP 4)		

4	Procs (OSPP 2), threads		
5	WS1	Thu, 2017-02-09 Mon, 2017-02-13	
6	Synchronization (OSPP 5)		
6	WS2	Thu, 2017-02-23 Mon, 2017-02-27	
6	P1-prep	Thu, 2017-02-23 Sun, 2017-02-26	Fri, 2017-03-03 23:59
7	Scheduling (OSPP 7.1, 7.2)		
7	WS3	TTh, 2017-02-28/03-02 Mon, 2017-03-06	
7	P1	Mon, 2017-02-27, 00:00	Sun, 2017-03-12, 23:59 Sun, 2017-03-19, 23:59
9	P2-prep	Wed, 2017-03-13, 00:00 Mon, 2017-03-20, 00:00	Fri, 2017-03-31, 23:59
10	**Spring break**		
11	Processes (OSPP 2.1, 2.7, 4.2) Kernel/user mode (OSPP 2.2-5) System calls (OSPP 2.3-3/6-7) Prog. interface (OSPP 3.1-7) Addr translation (OSPP 8.1-3) Virtual memory (OSPP 9) File systems (OSPP 11, 12)		
11	WS4	TTh, 2017-03-28/30 Mon, 2017-03-27	
11	P2	Mon, 2017-03-27, 00:00	Wed, 2017-04-12, 23:59
12	WS5	TTh, 2017-04-04 Mon, 2017-04-03	
14	[OSPP TBD]		
14	WS6		
14	P3-prep	Mon, 2017-04-17, 00:00	Fri, 2017-04-21, 23:59
14	P3	Mon, 2017-04-17, 00:00	Sun, 2017-05-07, 23:59

17	**Final**	Mon, 2017-05-08, 00:00	Thu, 2017-05-14, 23:59

Note: *Local links in table cells don't work. Use the Table of Contents to navigate.*

Workshop 1

Prereqs

OSPP: 2, 4, 5
Pintos: 1, 2, A

Tasks

UCD	MSUD	Task
black	yellow	Kernel initialization: basic steps
gray	blue	Interrupt system: setup and operation
cardinal	red	Face frames: what they are, why they are needed, how they work
white	orange	Context switch: mechanics and implications
violet	indigo	Queues (aka lists): what they are, implementation, API
amber	green	Synchronization: what it is, why it's used, how it's used

Workshop 2

Prereqs

OSPP: 2, 4, 5, 7.1, 7.2
Pintos: 1, 2, A

Tasks

UCD	MSUD	Task
cardinal	red	Semaphores, lists: APIs and usage details
amber	green	Scheduling fundamentals

black	yellow	Optimal scheduling
gray	blue	P1 - Alarm clock: problem, files. tests
white	orange	P1 - Priority scheduling: problem, files, tests
violet	indigo	P1 - MLFQS: problem, files, tests

Assignment P1-prep

Team or individual

Team

Requirements

1. Submit on Slack a *min 2-page* PDF document on your topic from WS2.
2. Document *quality* should be comparable to the Pintos Manual:
    a. Succinct.
    b. Full sentences.
    c. Clear.
    d. Sufficient.
    e. Minimal examples in formatted code.
3. Document structure (approximate):
    a. Overview
    b. *Detailed but brief* discussion (main points, sources, example code, figures).
    c. Open questions.
    d. Conclusion.
4. Code should be formatted and syntax highlighted. For Google Docs, use the Code Pretty addon. For other authoring packages, use the equivalent.
5. Think of this document as a *minimal reference* on a topic or assignment section. Any of your peers who was not on your team should be able to confidently start work on a task or topic using your document.
6. Take inspiration and guidance from the P1 Design Document, which will be part of your submission of P1.

Submission

On Slack, by mentioning me in your channel.

Graded

Yes, on scale {1, 2, 4, 5}.

No threes

There are is no 3, just as we don't have C/D grades. If your work is mediocre, it will get a 2. 4 is very good or excellent. 5 is outstanding and you might get a bonus or extra credit, at the instructor's discretion.

Assignment P1

Team or individual

Team

Requirements

1. The requirements are in the Pintos Manual, Chapter 2.

Submission

Submission is by ***zip*** or ***tar.gz*** archive containing:

1. *pintos/src/** and *pintos/tests/**
2. The Design Document in the top directory of the archive. The front page of the document should contain:
    a. Your team name
    b. Your teammates' names and emails
    c. The **number of tests passed** and a list of them with pass/FAIL (copy/paste from the output of *make check*)

Graded

Yes, by score, on:

1. Number of tests passed, out of a total of 27.
2. Code quality, on scale {1, 2, 4, 5}. No 3-s. See here.
3. No plagiarism and no unacknowledged or undescribed code, on *Pass/Fail* basis.

Max score is, therefore, 27+5=**32**, and min score is **0**.

Setup

You have two options to set up your Pintos programming environment:

1. Follow the instructions in (1) and (2) at the OS Playground.
2. Download an already configured OS image and open/install in VirtualBox. The image size is <3GB. Note that you need to install VirtualBox first. Follow the beginning of the instructions in (1) at the OS Playground. When you have VB installed, go the **File** menu, click **Import Appliance…**, and select the downloaded image. Follow the instructions.

Guidance

A few pointers on the assignment:

1. **Building Pintos.** You build/rebuild the kernel by running on the command line in a terminal the *make* command in *src/threads/build* and you tear it down by running *make clean*. Note that this command will remove your test results as well.
2. **Github.** You can integrate your Github repository with your Slack channel. Repository events (e.g. commits with comments) will appear in your channel. **NOTE:** Please, do not publish solutions code in public repositories. This degrades the value of Pintos as an educational resource. Students are eligible for unlimited free *private* repositories.
3. **Order of implementation.** Follow the order of the assignment sections: alarm clock, priority scheduling, advanced scheduler. They ramp up in this order.
4. **Splitting the work.** Split the work but check in with the team often, say, at least once a day. Things you can split on: alarm clock, priority use cases, priority donation cases, fixed-point data type, MLFQS statistics.
5. **Tests.** Tests are your *de facto* requirements. Run the tests. How do you run all tests? How do you run a single test? How do you run only some tests? What does the running of a test actually do? Read the tests. What does a test actually test? What determines if a test is PASS or FAIL? Make sure you understand the output of the *diff* command.
6. **Thread lifecycle.** Draw the thread lifecycle of the thread: creation —> {READY, RUNNING, BLOCKED, DYING}. Identify all transitions, especially among RDY, RUN, BLK. Identify the exact bare-Pintos code that affects each transition. Identify what transitions each test is testing, and, from this, design the changes and additions. The priority and priority-donation cases are best examined in the context of the thread lifecycle.
7. **Ordered lists.** You will need three types of ordered lists: sleeping threads, priority queues (e.g. ready list), and priority donors. Check out the following *function type* alias and its uses in **list.h**:

```
/* Compares the value of two list elements A and B, given
   auxiliary data AUX.  Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool list_less_func (const struct list_elem *a,
                             const struct list_elem *b,
```

```
                            void *aux);

/* Operations on lists with ordered elements. */
void list_sort (struct list *,
                list_less_func *, void *aux);
void list_insert_ordered (struct list *, struct list_elem *,
                          list_less_func *, void *aux);
void list_unique (struct list *, struct list *duplicates,
                  list_less_func *, void *aux);

/* Max and min. */
struct list_elem *list_max (struct list *, list_less_func *, void *aux);
struct list_elem *list_min (struct list *, list_less_func *, void *aux);
```

8. **Global/shared data.** All the queues you will be accessing are either global (e.g. ready queue) or shared (e.g. donors). You need to write them with **interrupts disabled**. This, in turn, puts constraints on the overhead you can afford for these operations. Note that if you overdo the interrupt disablement your tests might start failing in weird ways.
9. **Alarm clock.** There are two operations: putting to sleep and waking. One is done in thread context, the other in interrupt context. Interrupt handlers are **not** threads so you have to use a semaphore (semaphores have no owner so they can be *signaled/upped* in any context) instead of a lock (locks have owning threads and only the owner can *release* them). What state is *sleeping* in the context of the thread lifecycle? How many semaphores do you need for *n* sleeping threads?
10. **Timer interrupt.** If you find that you need to run the scheduler after the completion of an interrupt handler, check out the following function in interrupt.c and find out how it's used in the bare-Pintos round-robin scheduler:

```
/* During processing of an external interrupt, directs the
   interrupt handler to yield to a new process just before
   returning from the interrupt.  May not be called at any other
   time. */
void
intr_yield_on_return (void)
{
  ASSERT (intr_context ());
  yield_on_return = true;
}
```

11. **Fixed-point number.** The implementation asks you to use an *int* but interpret the bit pattern as a fixed-point number. Wrap the raw *int* in a *struct* to protect it from incorrect interpretation and conversion errors.
12. **Expectations.** Test your expectations for the arguments of functions and other conditions throughout your code. Use the ASSERT macro from **lib/debug.h**:

```
/* This is outside the header guard so that debug.h may be
```

```
   included multiple times with different settings of NDEBUG. */
#undef ASSERT
#undef NOT_REACHED

#ifndef NDEBUG
#define ASSERT(CONDITION)                                       \
        if (CONDITION) { } else {                               \
                PANIC ("assertion `%s' failed.", #CONDITION);   \
        }
#define NOT_REACHED() PANIC ("executed an unreachable statement");
#else
#define ASSERT(CONDITION) ((void) 0)
#define NOT_REACHED() for (;;)
#endif /* lib/debug.h */
```

13. **Priority donation.** List the things that need to happen upon priority donation. How many priority values should a thread keep track of? Which of them is modified when? Think of the priority donation cases in the context of the thread lifecycle. Read the tests closely.
14. **Recursion.** Priority donation is *recursive*. Think of what this means and how much work you need to do upon priority a donation cascade. Make sure you recurse correctly. There are incorrect ways to set up your recursion. Make sure you recalculate priorities after a donation propagates all the way and before you re-enable interrupts.

Workshop 3

Prereqs

OSPP: 2, 4, 5, 7.1, 7.2
Pintos: 1, 2, A

Tasks

UCD	MSUD	Task
gray	blue	Running the Pintos tests. C code. Expected output.
black	yellow	Synchronization API at different levels of abstraction.
amber	green	Priority donation cases.
cardinal	red	Ownership in semaphores and locks in Pintos. Implications.
white	orange	Atomic two-operation instructions. x86.

violet	indigo	Linux scheduler. Before and after 2007.

Workshop 4

Prereqs

OSPP: **2**, **3**, 4, 5, 7.1-5, **8.1-3**, **9**, 11, 12
Pintos: **3**, **A.4.2-A.7.3**

Tasks

UCD	MSUD	Task
gray	green	P2 - Argument passing: problem, files, tests
black	yellow	P2 - Process control syscalls: problem, files, tests
amber	orange	P2 - File operation syscalls: problem, files, tests
cardinal	red	System calls: Overview, mode transfer, `INT` instr., internal interrupt
white	(orange)	Virtual memory: Overview, addr translation, Pintos VM
violet	(red)	File systems: Overview, abstraction stacks (p.506), Pintos FS

Assignment P2-prep

Team or individual

Team

Requirements

1. Submit on Slack a *min 2-page* PDF document on your topic from WS4.
2. Document *quality* should be comparable to the Pintos Manual:
    a. Succinct.
    b. Full sentences.

    c. Clear.
    d. Sufficient.
    e. Minimal examples in formatted code.
3. Document structure (approximate):
    a. Overview
    b. *Detailed but brief* discussion (main points, sources, example code, figures).
    c. Discussion.
    d. Open questions.
4. Code should be formatted and syntax-highlighted. For Google Docs, use the Code Pretty addon. For other authoring packages, use the equivalent.
5. Think of this document as a *minimal reference* on a topic or assignment section. Any of your peers who was not on your team should be able to confidently start work on a task or topic using your document.
6. Take inspiration and guidance from the P2 Design Document, which will be part of your submission of P2.
7. Do a round-robin review a day before submission, to clean the document up with fresh eyes.

Submission

On Slack, by mentioning me in your channel.

Graded

Yes, on scale {1, 2, 4, 5}.

No threes

There are is no 3, just as we don't have C/D grades. If your work is mediocre, it will get a 2. 4 is very good or excellent. 5 is outstanding and you might get a bonus or extra credit, at the instructor's discretion.

Assignment P2

Team or individual

Team

Requirements

1. The requirements are in the Pintos Manual, Chapter 3.

Submission

Submission is by ***zip*** or ***tar.gz*** archive containing:

1. *pintos/src/** and *pintos/tests/**
2. The Design Document in the top directory of the archive. The front page of the document should contain:
    a. Your team name
    b. Your teammates' names and emails
    c. The **number of tests passed** and a list of them with pass/FAIL (copy/paste from the output of *make check*)

Graded

Yes, by score, on:

1. Number of tests passed, out of a total of 80.
2. Code quality, on scale {1, 2, 4, 5}. No 3-s. See here.
3. No plagiarism and no unacknowledged or undescribed code, on *Pass/Fail* basis.

Max score is, therefore, 80+5=**85**, and min score is **0**.

Setup

You have two options to set up your Pintos programming environment:

1. Follow the instructions in (1) and (2) at the OS Playground.
2. Download an already configured OS image and open/install in VirtualBox. The image size is <3GB. Note that you need to install VirtualBox first. Follow the beginning of the instructions in (1) at the OS Playground. When you have VB installed, go the **File** menu, click **Import Appliance…**, and select the downloaded image. Follow the instructions.
3. The rest are steps, specific for P2:
    a. IMPORTANT: If you continue developing on top of your P1 code, you need to first do `make clean` at the top pintos directory (`~/pintos` in the Ubuntu VM).
    b. Open `~/pintos/src/utils/pintos`. On line **257** change the `…/threads/...` part of the path to the kernel to `…/userprog/…`. Save the changes.
    c. Open `~/pintos/src/utils/Pintos.pm`. On line **362** change the `…/threads/...` part of the path to the kernel to `…/userprog/…`. Save the changes.
    d. In `~/pintos/src/userprog` do `make`. This should also create `~/pintos/src/userprog/build`.
    e. Change to the `build` subdirectory.
    f. Run `pintos-mkdisk filesys.dsk --filesys-size=2`. This creates a *simulated* disk in the same directory.

g. Run `pintos -f -q`. This will format the disk with a filesystem partition. NOTE: If you are getting a message from Pintos that there is no `-f` option, then you have missed one of steps {a, d}. Pintos has to be rebuilt under `userprog/` for the option to appear. Notice that there is a directory `userprog/build/filesys` now. This behavior is controlled by the `Makefile` variable `FILESYS`, which is not defined when Pintos is built under `threads/` but is defined for builds under `userprog/`, `vm/`, and `filesys/`. See `DEFINES` in `userprog/Make.vars`.
h. Before you can run `echo` from the examples, you need to run `make` in `~/pintos/src/examples`. Now you have an `echo` executable.
i. Now, back in `~/pintos/src/userprog/build`, you can run `pintos -p ../../examples/echo -a echo -- -q`. (Notice the double dash before `-q`.) This will load the `echo` executable into the filesystem. You can run `echo` with `pintos -q run 'echo x'`. **Note:** These are single quotes around `echo x`, not backticks.

Guidance

1. Process lifecycle.
    a. Lifecycle of parent and child. The following interdependencies should be taken into account:
        i. The parent creates the child. The creation may be successful or unsuccessful. If successful, the parent adds the child to its children. If unsuccessful, parent cleans up, if necessary.
        ii. If the child is successfully created, whether it exits before the parent returns from *thread_create* or before the parent calls *wait ()* on it or after any of these events, the parent should add the child to its list of children.
        iii. The child may exit after the parent returns from *thread_create ()* but before the parent calls *wait ()* on the child. In this case the parent will add the child to its children. Upon calling *wait ()* on the child, it should block until the child is dead, and clean up.
        iv. The child may exit after the parent calls *wait ()* on the child. Same as the previous case.
        v. The parent can exit before the child. In this case the child should clean up.
        vi. So, there are three important cases:
            1. Both parent and child still alive
            2. One of them is dead, the other alive
            3. Both parent and child are dead
        vii. **Note:** The cases above do not necessarily map straightforwardly to the thread states *READY, RUNNING, BLOCKED, DYING*.
    b. Stacks.

i. Between the parent and the child, there are 2 (kernel) threads and 3 stacks.
ii. The parent and child each have their usual kernel thread stack, which is in kernel address space.
iii. The child also sets up a second stack, this one on a page in user address space, to execute the user program.
iv. **Note:** Whenever data has to be copied from a kernel stack to a user stack or vice versa, the addresses have to be converted. There are two main cases:
    1. When program arguments are written from the child kernel stack to the newly created user stack. In this case, use the function *put_user ()* on p. 26 of the Manual.
    2. When system call arguments have to be copied from the user stack to the kernel stack. In this case, use the function *get_user ()* on p. 26 of the Manual.

c. Management of the executable file.
    i. The user program is contained in an executable file that the child process opens. It extracts the filename from the command line string (containing the file name and any arguments) and opens the file.
    ii. This is the file that is being loaded in *load ()*. Loading means that the file is read, verified, and its executable and data segments are copied over to user space pages created for the process. See the diagram on p.25 of the Pintos Manual.
    iii. The base Pintos system, that is, before starting P2, does two things that shouldn't be done in P2:
        1. It makes a copy of the filename to avoid a race condition between the parent and child threads. This is eliminated with a semaphore synchronizing the two threads on the event of successful load of the child.
        2. It closes the executable file at the end of *load ()*. The file should not be closed while the child is alive. See the end of section **3.3.5** in the Manual. The child thread should hold on to it while it is executing. This requires a new variable in *struct thread*.

d. Process creation.
    i. *process_execute ()* is in the invoking (parent) thread.
    ii. *start_process ()* is in the newly created (child) thread.
    iii. At the end of *start_process ()*, the child starts executing the user program code on the process stack. This is the effect of the last line of the function which uses assembly to mimic a "return from interrupt" after having set up the pointer to the next instruction (*eip*) to the be the first instruction of the user program and the stack pointer (*esp*) to be at the top of the user stack. Both of these are in the "fake" stack frame that was created in the

stack frame of `start_process` in the kernel stack for the mimicry to work.

iv. There are two structures containing data shared between the parent and the child that have different roles. The first one is used by the parent to pass the necessary data (command line and a pointer) to the child and get timely feedback on the success of its loading and process setup. Let's call it *struct child_exec*. It is declared locally in *process_execute ()* and should contain:

1. A boolean success variable: The parent needs to know if the child was created successfully, so it can add it to its list of children (see further down), or else clean up and report error in *process_execute()*! The parent initializes it to *false*, and the child may set it to *true* if everything went okay right before it ups the *child_loaded* semaphore (see below).
2. Command line, containing the executable file name and any program arguments.
3. A semaphore *child_loaded* to be used for synchronization on the successful loading of the child. It is initialized to 0 and downed by the parent. It is upped by the child at the very end of *start_process ()* if everything went okay.
4. A pointer to a second structure containing more data shared between the child and the parent (see below). It is initialized to *NULL* by the parent and is allocated dynamically by the child in *start_process ()*. This pointer makes it possible for both parent and child to have access to the same structure.

v. **Note:** You can only pass one address via the aux argument of *thread_create ()*! In C, a structure is defined and allocated, and then a pointer to it is passed as the single argument. That way, you get access to multiple data points. Remember to create a local structure pointer to which to assign the passed-in pointer, so you can access the members of the structure the usual way: *s->file_name* or *sema_down (s->some_semaphore)*.

e. Children.

i. So thread structure has both a child element, and a list of children. The child element is in the dynamically allocated *child_parent* structure. The list of children in in the thread structure directly.

ii. **Note:** A process has only one parent.

iii. **Note:** The system call *wait ()* cannot be called on grandchildren.

f. The second structure, call it *struct child_parent*, is allocated dynamically (that is, using *malloc*) by the child in *start_process ()* and assigned to the pointer in the *child_exec* structure passed in by the parent. Note that *malloc* allocates memory dynamically on a heap, not on any thread's stack. This is important for the ability of either parent or child to free it. The structure should contain:

i. A child list elem of type *struct list_elem*, which the parent will use to add the child to its children list.
ii. A integer reference count used to distinguish among three cases:
    1. 2 ⇒ both parent and child are alive
    2. 1 ⇒ only one of them is alive
    3. 0 ⇒ both are dead
iii. These cases take care of the different scenarios that come up between the parent and child (see Lifecycles above).
iv. The reference count is initialized to 2 by the child, meaning that, to the best of its knowledge, both parent and child are alive. When either parent or child exits, or dies for some other reason, it should decrement the reference count. If, after decrementing, the reference count is 0, the thread should free the *child_parent* structure.
v. A lock to protect reading and writing of the reference count. Both parent and child have to check and/or modify the reference count, so remember to use *acquire* and *release* around each access.
vi. The child's thread id so the parent can search efficiently upon a *wait()* system call.
vii. An integer exit code for the child, so the parent can know how the child exited.
viii. A semaphore *child_dead* to synchronize the parent and the child on the dying of the child in the *wait()* system call. The semaphore is initialized by the child to 0 in *start_process*. Upon *wait ()*, the parent downs the semaphore. The child ups it upon dying.
ix. **Note:** The down and up of a semaphore initialized to 0 can happen in *either order*. Do not assume any particular order and don't base your decision of which thread should free the *child_parent* structure. Work with the reference count for a consistent approach.

2. Arguments.
    a. User program.
        i. In terms of the process creation and execution sequence, the arguments have to be set up right after the creation of the stack. This means you need to percolate the command line all the way to *setup_stack ()*, for which you will need to add another argument.
        ii. Obviously, you can only set up the stack **after** and upon success of the process address space mapping of the page in *install_page ()*. This is the place to pass the arguments. The best description of the actual procedure can be found in section **3.5.1** on pp.35-36 of the Pintos Manual.
        iii. If you want to use *hex_dump ()* to check the contents of the memory you just wrote, as is suggested in the Manual, you do it by using *hex_dump ()* the way you would use *printf ()*, that is, add *hex_dump ()* lines directly in your code after some operation on a memory object to see the actual bytes, optionally rendered in ASCII (last `bool` argument). The output is to

the console. See the documentation in *src/lib/stdio.c* and examples in *src/filesys/fsutil.c*, *src/lib/kernel/bitmap.c*, and *src/tests/lib.c*.

iv. **Note:** You are copying **from** *kernel address space* (you are still executing in the address space of the kernel thread) **to** *user/process address space* (the process stack at the top of the just mapped page). Use the *put_user ()* function on p.26 in the Manual. (The assembly code inside this function essentially passes a memory address in user address space through the MMU, which translates it to the mapped physical address for this process, and writes the passed value at that address.)

v. It's best to encapsulate the parsing and putting of the arguments onto the process stack in a separate procedure.

vi. **Note:** The *esp* argument to *setup_stack ()* is used to **return** the updated stack pointer. This is an old C technique: by using a pointer to an external value, one can let the function modify it, and, effectively, get an extra "return" value. This can be done with any number of arguments. In the case of *setup_stack ()*, since the variable (*esp*) is already a pointer, we need a double pointer (*void **esp*) to use this technique

b. System call.

i. The user stub pushes the arguments in reverse order on the user stack, then the system call number. The the *INT 0x30* instruction is executed which triggers the interrupt mechanism for a system call. The system switches from ring 3 (user mode) to ring 0 (kernel mode) and starts executing on the kernel thread's stack.

ii. As usual, the interrupt mechanism creates a interrupt frame on the user stack and populates it with the register state of the process, so that it can be restored and resumed properly when the system returns from the system call interrupt. The most important registers are:

1. EIP - the next instruction of the user program that will be executed
2. ESP - the process (user) stack pointer
3. EAX - the register where to put the return value of the system call, whatever it is

iii. The *syscall_handler ()* receives a pointer to the interrupt frame as an argument. ESP will point to the top of the user/process stack where the number of the system call is.

iv. The *syscall_handler ()* is executing on the kernel thread stack, so all addresses from the user/kernel, including the ESP, have to be translated from user virtual address space to kernel address space, using the *get_user ()* function on p.26 in the Manual.

3. Memory.

a. Physical memory.

i. The physical memory is divided into 4K frames, where 4K pages of virtual memory are mapped. (See Virtual memory below) The mapping allows

memory pages to be swapped into physical memory for execution and out for storage while not needed.

b. Virtual memory.
    i. Virtual memory is an abstraction used to create the illusion of infinite memory to the processes running on an operating system. The principal element of the abstraction is the address space. (See Address spaces below).
    ii. Kernel is global and doesn't change, process is process-specific and changes. **Note:** This means that different kernel threads are always going to have different non-overlapping address spaces.
    iii. Kernel pages are mapped one-to-one to physical frames, 3GB-4GB, while each process has pages that are mapped to the region of 128MB-3GB. **Note:** This means that different user processes have overlapping address spaces since they start and end at the same address (that is, the start of the downward growing stack)!
    iv. Page directories make sure overlapping user address spaces are actually mapped to non-overlapping kernel pages and physical frames.
    v. Note: The byte at *PHYS_BASE* is part of kernel memory. Setting the process stack pointer *esp* to *PHYS_BASE* means the stack is empty. The process stack grows downward.
c. Address spaces.
    i. IN PROGRESS
d. Memory management.
    i. Management of page directories.
        1. Activation, deactivation.
    ii. Allocating and deallocating pages.
    iii. Copying between user and kernel pages.
        1. Note: User pages are mapped to physical pages via kernel pages!
    iv. Bad addresses.
        1. IN PROGRESS

4. File lifecycle.
    a. IN PROGRESS
5. System calls.
    a. Interrupt.
        i. System calls pass through a "narrow gate" or "funnel neck". This is the **INT 0x30** instruction in the user stub.
        ii. The user stub is on the user side. Make sure you understand the use of the user stack for system call arguments.

Workshop 5

Prereqs

OSPP: **2**, **3**, 4, 5, 7.1-5, **8.1-3**, **9**, 11, 12
Pintos: **3**, **A.4.2-A.7.3**
Files: All files mentioned in the designated manual reading. Additional suggestions:

Tasks

UCD	MSUD	Task
gray	green	Process lifecycle
black	yellow	Arguments
amber	orange	Memory
cardinal	red	File lifecycle
white		System calls
violet		Stacks

Assignment P3-prep

Team or individual

Team

Requirements

8. Submit on Slack a *min 2-page* PDF document on your topic from WS6.
9. Document *quality* should be comparable to the Pintos Manual:
    a. Succinct.
    b. Full sentences.
    c. Clear.
    d. Sufficient.

    e. Minimal examples in formatted code.
10. Document structure (approximate):
    a. Overview
    b. *Detailed but brief* discussion (main points, sources, example code, figures).
    c. Discussion.
    d. Open questions.
11. Code should be formatted and syntax-highlighted. For Google Docs, use the Code Pretty addon. For other authoring packages, use the equivalent.
12. Think of this document as a *minimal reference* on a topic or assignment section. Any of your peers who was not on your team should be able to confidently start work on a task or topic using your document.
13. Take inspiration and guidance from the P3 Design Document, which will be part of your submission of P3.
14. Do a round-robin review a day before submission, to clean the document up with fresh eyes.

Submission

On Slack, by mentioning me in your channel.

Graded

Yes, on scale {1, 2, 4, 5}.

No threes

There are is no 3, just as we don't have C/D grades. If your work is mediocre, it will get a 2. 4 is very good or excellent. 5 is outstanding and you might get a bonus or extra credit, at the instructor's discretion.

Assignment P3

Team or individual

Team

Requirements

1. The requirements are in the Pintos Manual, Chapter 4.
2. Note that Section 4.1.1 has changed from the Stanford version of the Manual. The text that corresponds to the current Pintos codebase is as follows:

You will work in the `'vm'` directory for this project. The `'vm'` directory contains only `'Makefile'`s. The only change from `'userprog'` is that this new `'Makefile'` turns on the setting `'-DVM'`. All code you write will be in new files or in files introduced in earlier projects.

You will probably be encountering just a few files for the first time:

`'devices/block.h'`
`'devices/block.c'`

> Provides sector-based read and write access to block device. You will use this interface to access the swap partition as a block device.

Submission

Submission is by ***zip*** or ***tar.gz*** archive containing:

1. *pintos/src/** and *pintos/tests/**
2. The Design Document in the top directory of the archive. The front page of the document should contain:
    a. Your team name
    b. Your teammates' names and emails
    c. The **number of tests passed** and a list of them with pass/FAIL (copy/paste from the output of *make check*)

Graded

Yes, by score, on:

1. Number of tests passed, out of a total of 113, some of which are regression tests from P2..
2. Code quality, on scale {1, 2, 4, 5}. No 3-s. See here.
3. No plagiarism and no unacknowledged or undescribed code, on *Pass/Fail* basis.

Max score is, therefore, 113+5=**118**, and min score is **0**.

Tests

For P3, there are the following tests:

1. 34 functionality (new) tests
2. 66+13=79 regression (robustness) tests from P2.

Follow this link for a detailed view: https://goo.gl/M2fFvO.

Note that temporarily you will break some of the tests that you passed for P2 before they pass with the new functionality. This is okay. Don't be afraid to break passing tests. On the other hand, it's best to get them to pass again asap. This means that you should work in increments of adding new functionality and fixing the corresponding tests (as much as possible) before adding more functionality. Don't go out on a limb and break everything. Stay close to familiar behavior.

Setup

You have two options to set up your Pintos programming environment:

1. P3 is based on P2. You should build it on top of your code from P2. If you are not passing some tests from P2, you should first get them to pass. P3 runs some of the P2 tests as regression tests.
2. Do `make clean` at the top pintos directory (`~/pintos` in the Ubuntu VM).
3. The rest are steps, specific for P3:
    a. Open `~/pintos/src/utils/pintos`. On line **257** change the `…/userprog/...` part of the path to the kernel to `…/vm/…`. Save the changes.
    b. Open `~/pintos/src/utils/Pintos.pm`. On line **362** change the `…/userprog/...` part of the path to the kernel to `…/vm/…`. Save the changes.
    c. In `~/pintos/src/vm` do `make`. This should also create `~/pintos/src/vm/build`.
    d. Change to the `build` subdirectory.
    e. Run `pintos-mkdisk filesys.dsk --filesys-size=2`. This creates a *simulated* disk in the same directory.
    f. Run `pintos -f -q`. This will format the disk with a filesystem partition.

GUIDANCE

How to use this guidance

This guidance is not meant to substitute reading of the Pintos Manual. The explicit requirements are in the manual, even though they may be somewhat scattered. This guidance tries to fill in the blanks and impose a more rigid structure and consistent approach to the design and implementation of the major functional components of the project. The best way to use it, though, is side-by-side with the Pintos Manual and your own team designs.

Resources

Godmar Back's VM handout: https://goo.gl/1On6yV
Godmar Back's P3 guidelines: https://goo.gl/z85D9j

General

1. VM revisited

The virtual memory (VM) infrastructure for P2 was, like the filesystem, minimal, just enough to support processes making system calls. P3 will build it out with three major features:

- User page allocator
- Stack growth
- Memory-mapped files

2. User pages

The P2 VM was based on a piggybacking mechanism of automatic mapping of user pages to kernel pages, encapsulated in the *install_page* function. Since Pintos kernel pages are mapped one-to-one to physical memory frames, by mapping a user page to a kernel page, we got it also mapped to a physical frame. In P3 we are not going to piggyback on this Pintos feature but we are going to map our process (user) pages and manage them through their lifecycle ourselves.

3. User page allocator

The user pages have a rich lifecycle: they are allocated, mapped to frames, evicted, swapped out, swapped in, deallocated, mapped to files, and destroyed. Throughout this lifecycle, information about each process page has to be tracked and kept updated. This is the major complexity of the project. Pages are per process, frames are global, mappings to files are per process.

4. Frames

Frames are page-sized regions of physical memory. To have a process execute, it has to have at least one page in physical memory. Since pages can be mapped to any available frame (that is, not already mapped), the mapping is dynamic (that is, not fixed over the whole lifetime of the process and its pages). Again, pages are per process, but frames are global. Anything that needs to execute has to be in a page that is in a frame.

5. Frame eviction

There are only so many frames and the demand for pages required by all processes that are running can surpass the number of frames. In this case, pages in frames have to be evicted to make room for pages that are not but are needed by their processes. This is also a dynamic process. The eviction policy has to be efficient in the sense of minimizing the swapping of pages in and out. Swapping incurs an overhead just like context switching for threads. We don’t want to evict a page that is likely to be wanted back right away.

6. Swap space

When a page is evicted, it has to be saved somewhere in case and until it is again needed for execution. This is where the block-device based swap space comes in. It simulates a special *swap* partition in secondary storage (that is, a disk). Evicted pages are saved to the swap partition (swapped out) and returned to physical (main) memory (swapped in) when they are needed. This is also a dynamic mapping that has to be kept track of.

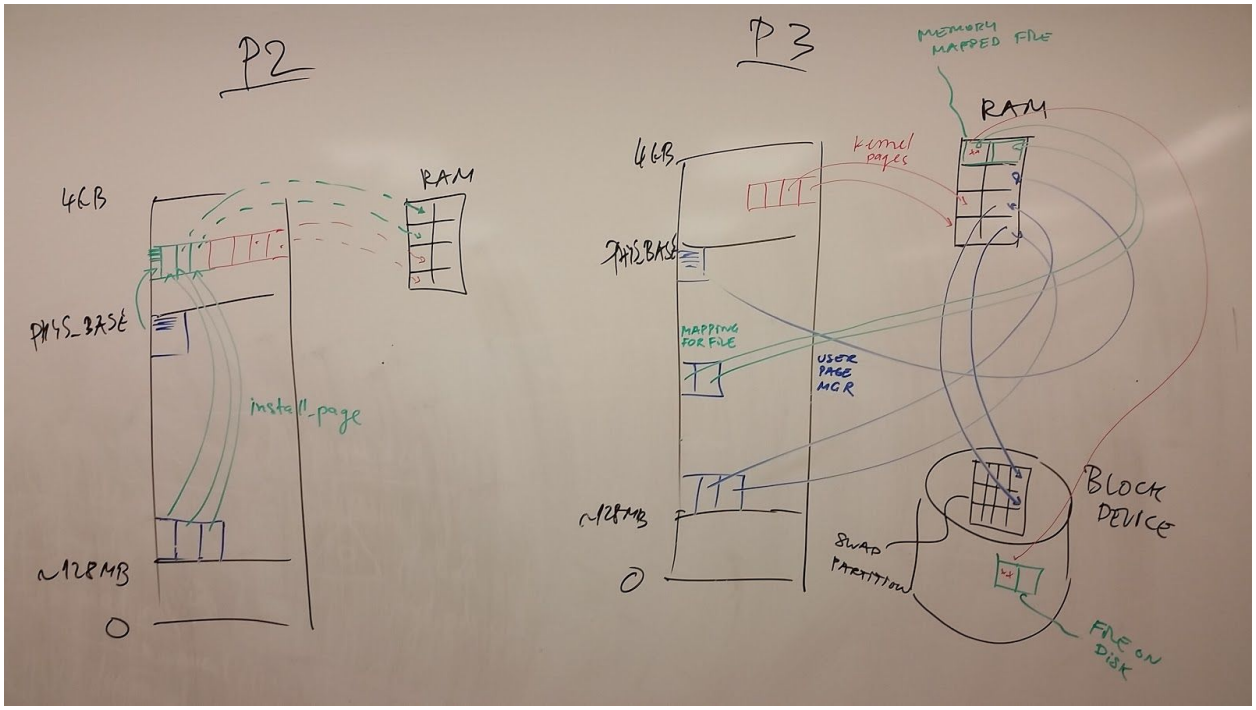
7. Memory mapped files

When processes need to work on files, these files also have to be in physical memory. The process needs to know which frames are occupied by the file. The file also needs to be in user space for the user program (process) to work on it, so the process maps the necessary number of pages to the file frames. If multiple processes need to work on the same file, we don't want to bring it into memory multiple times (unless that is expressly necessary). Each process maps its pages to the same file frames.

8. Synchronization

Since our filesystem is still primitive, we continue to use the global filesystem lock. There is also locking for pages operations (per process) and frame operations (global). Naturally, there is synchronization for memory-mapped files.

Sketch



Virtual memory in P2 and P3.

Note: Not quite accurate. Here's a clarification until I can draw another:

1. We never quite touch the RAM frames directly. All access to RAM is controlled by the OS at a very deep level and all we can do is read/write to *kernel* or *user address space*.
2. The kernel address space is mapped one-to-one to RAM frames and that's how we get to write directly to the *frames*. Notice that all the RAM pages - that is, for both *kernel* and *user* pools - are in the ***kernel address space***!
3. In P2, we used `install_page` to immediately map any page in the user address space we needed for the process (these were the pages for the executable and data

segments and the single stack page) to the ***user pool of the kernel address space***. *(Are you getting a headache or what?!)*

4. What we were missing in P2 was direct control over the mapping, demand-paging (aka lazy paging), eviction, and memory-mapped files. All of these are implemented in P3 by first decoupling the user page allocation from mapping to a frame. We abandon `install_page` and the in-advance loading of the page contents and substitute with anticipatory `page_allocate` and `page_fault`-triggered `page_in`.

Tests

Some notes on the P3 tests can be found here.

Notes

Some general notes on P3 can be found here.

Specific guidance overview

Topics are:

1. Frames.
2. Pages.
3. Swap partition.
4. Memory mapped files.
5. Second chance algorithm.

Most are organized into:

1. Overview.
2. Files.
3. Data structures.
4. Lifecycle.

Frames

Overview

By “frames”, we will refer to two distinct things, which, if not obvious from the context, will be qualified:

1. A (physical) frame is a page-sized page-aligned region of physical memory (aka main memory, aka RAM). **Note:** A page (see Pages) needs to be loaded into a frame to be executable.
2. A frame is a `struct frame` data structure which holds the necessary data for our frame management.

Physical frames represent the hardware RAM available to Pintos, and so the array of frame structures is **global** (that is, defined as a global in `vm/frame.c`).

Files

`vm/frame.h`

- `struct frame`
- Frame management function declarations

`vm/frame.c`

- Global `frames` array
- Global `scan_lock` to protect `frames` traversal
- Global `frame_ct`, the number of physical frames represented
- Global `hand`, in a reference to a *clock hand*, for the *page eviction algorithm*
- Frame management function definitions

Data structures

```
/* A physical frame. */
struct frame
  {
    struct lock lock;           /* Prevent simultaneous access. */
    void *base;                 /* Kernel virtual base address. */
    struct page *page;          /* Mapped process page, if any. */
  };
```

Things to note:

- Access to each frame should be synchronized.
- Each frame has a kernel virtual (page) address, which, due to the one-to-one mapping of kernel pages to physical frames, coincides with the physical frame address.
- A frame is associated with a page (which is loaded in it), or not. If it is not, it is available. If it is, the page has to be examined for eviction according to the *page eviction policy/algorithm*; if the page can be evicted, the frame can become available.

Lifecycle

- **Initialization.** The frames should be initialized in the Pintos `main` function in `init.c`. The physical frames from the *user pool* are obtained with `palloc_get_page (PAL_USER)` until it returns `NULL`. A `struct frame` is allocated (with `malloc`) for each physical frame obtained from the page allocator. **Note:** You will no longer call `palloc_get_page ()` in `load ()` and `setup_stack ()`, but you will call the corresponding function of your own page allocator to allocate a page (see Pages).
- **Locking for paging.** When the page manager is manipulating a page, it has to lock the page's frame, if it has one. The `struct frame`'s lock is used. *Hint:* It is good to encapsulate the necessary checks and asserts in `frame_lock` and `frame_unlock`. These functions will take a (pointer to a) `page struct` as argument. A frame should not be able to change (contents or status) between the calls to these functions, which

means that only the holder of the lock (which should be the current thread) can make changes. If the page in the argument has no frame, do nothing.

- **Locking for scanning.** When the physical frames are scanned for whatever reason (e.g to find an available frame (to allocate) for a new page) the global `scan_lock` should be held.
- **Allocation.** Frame management is done by the page manager, that is, the page manager calls the frame management functions. The page manager requests the allocation of a frame for a page. (**Note:** The physical frames already exist, and the struct frame-s are already allocated, so "allocation" here means the atomic association of a page with a frame. The actual writing of the frame with the page contents, called "paging in", is done by the page manager when a frame has been allocated for the page.) Allocation of a frame to a page means setting `page` in the `struct frame`. Before trying any eviction, the `frames` array should be traversed to look for an available frame. If none is, the eviction algorithm should kick in. The function(s) for allocating a frame should return (a pointer to a) `struct frame` or `NULL`. **Note:** When examining a frame for allocation, its lock should be held. If the lock cannot be obtained, that frame should be ignored. If none can be obtained, the thread can use `timer_sleep` from P1 to wait and try again.
- **Eviction algorithm.** The eviction algorithm is triggered when no free frame can be found. It treats the `frames` array as a circular sequence. (*Hint:* Use the modulo operator.) The global `hand` points to the next frame to try. It it's available (i.e. `page` is `NULL`), take it. If it is not, check if its page has been accessed recently (see Pages). If it was, advance the `hand`. If it wasn't, evict it. Note: The this check for recently accessed should clear the recent access flag `PTE_A` in the corresponding page table entry (see Pages). Therefore, the hand should complete **2** full turns around the clock in search for a page to evict. For this reason, this eviction algorithm is also known as *second-chance*.
- **Freeing.** A frame is freed by setting its `page` to NULL.

Pages

Overview

By "pages", we will refer to three distinct things, which, if not obvious from context, will be qualified:

- A page is a page-sized (`PGSIZE`) page-aligned (starting address is `0x*****000`) region in an (abstract) memory address space.
- A page is a page-sized page-aligned chunk of code or data that belongs to a process, that might be in a file (see Memory mapped files), frame (see Frames), or swap device (see Swap partition).
- A page is a `struct page` data structure which holds the data we need for page management.

There are two pools from which pages can be taken, *kernel* and *user*, and they are both managed by palloc and distinguished by the `PAL_USER` flag. In P2, page management was

done for us through the `install_page ()` mapper, which piggybacked on the one-to-one mapping of kernel pages to physical frames. In P3, we write our own page manager to map between user pages and physical frames (see Frames) **directly**. **Note:** The physical pages are already obtained for use by the page manager (see Frames) and so it does not need to ever call `palloc_get_page ()` itself.

Pages are **per process**. The process (supplemental) page table, managed by the user page manager, is a *hash table* which holds the `struct page` structures for pages allocated for the process.

The page manager/allocator will now have to handle page faults in user space. Page faults will therefore no longer indicate bugs, just pages which, due to lazy loading, are currently missing. *Lazy loading* means that a page is not allocated and paged into a frame until the first access of an address in this page. All pages can be loaded lazily, except the first user stack page which can be loaded in `setup_stack ()`.

Files

`vm/page.h`

- `struct page`
- User page management function declarations
- Declaration of a `hash_hash_func` for the `pages` hash table
- Declaration of a `hash_less_func` for the `pages` hash table

`vm/page.c`

- `STACK_MAX` (e.g. 1M)
- User page management function definitions

Data structures

```
/* A kernel thread or user process. Note: Partial. */
struct thread
  {
    /* Note: Other members here. */

    /* Owned by process.c. */
    int exit_code;                      /* Exit code. */
    struct wait_status *child_parent;   /* Shared data for parent and child. */
    struct list children;               /* (Shared data structs of) children. */

    /* Alarm clock. */
    int64_t wakeup_time;                /* Time to wake this thread up. */
    struct list_elem timer_elem;        /* Element in timer_wait_list. */
    struct semaphore timer_sema;        /* Semaphore. */

    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
```

```
    struct hash *pages;                 /* Page table. */
    struct file *bin_file;              /* The binary executable. */

    /* Owned by syscall.c. */
    struct list fds;                    /* List of file descriptors. */

    struct list mappings;               /* Memory-mapped files. */
    int next_handle;                    /* Next handle value. */

    void *user_esp;                     /* User's stack pointer. */

    /* Owned by thread.c. */
    unsigned magic;
  };
```

Things to note:

- The thread's page table is a hash table `struct hash`, implemented in `lib/kernel/hash.c` The actual `hash_elem` is in `struct page` for each page.
- Each process has its own file mappings which are based on file handles (see Memory mapped files).

```
/* Virtual page. */
struct page
  {
    /* Immutable members. */
    void *addr;                 /* User virtual address. */
    bool read_only;             /* Read-only page? */
    struct thread *thread;      /* Owning thread. */

    /* Accessed only in owning process context. */
    struct hash_elem hash_elem; /* struct thread `pages' hash element. */

    /* Set only in owning process context with frame->lock held.
       Cleared only with scan_lock and frame->lock held. */
    struct frame *frame;        /* Page frame. */

    /* Swap information, protected by frame->lock. */
    block_sector_t sector;       /* Starting sector of swap area, or -1. */

    /* Memory-mapped file information, protected by frame->lock. */
    bool private;               /* False to write back to file,
                                   true to write back to swap. */
    struct file *file;          /* File. */
    off_t file_offset;          /* Offset in file. */
    off_t file_bytes;           /* Bytes to read/write, 1...PGSIZE. */
  };
```

Things to note:

- The `addr` is the `0x*****000` address of the page, that is, the *bottom* of the page. This is the address that is returned by `pg_round_down` for any address. *Hint:* It would be convenient if you build the page allocator around this address, by getting the page for a particular user address (which you might need to write to or read from).
- Some of the user pages allocated in `load ()` should be read-only!
- You need to hold a pointer to the owner `thread` so you can check if it is holding the lock for the frame you want to manipulate.
- A page is associated with (the kernel thread of) a process through the `hash_elem`.
- A page is associated with a frame through a pointer to its `struct frame`.
- A page is associate with a swap slot (see Swap partition) by the index of the first sector of the slot. **Note:** The swap partition is a block device. A block is made up of sectors. A sector is the minimal operable unit of the block device. Sectors in a block device are (uniquely) indexed. The page size is an even multiple of the sector size. So page slots in the block device of the swap partition are contiguous regions composed the number of sectors that fit in a page. See Swap partition.
- A page can be mapped to only one file (see Memory mapped files). **Note:** `page->file` can be used as a test to determine if (the contents of) the page should be read from a file (rather than a swapped back in or zeroed out, for a new blank page, e.g. for stack growth).
- A page can be `private` or not. Memory-mapped file pages are always non-private, so `private = false` for them. For other pages, `private = !read_only`.

Lifecycle

- **Initialization.** The hash table has to be initialized. You can do that anywhere but it's best to do it after process_activate () in load (). **Note:** This is a two-step process. First you need to malloc the pages hash and then call `hash_init` on it. `hash_init` requires the `hash_hash_func` and `hash_less_func` functions you need to define. For the hash function, you can just return the number of the page (*Hint:* Use `hash_entry` to get the page and then return `addr` right-shifted by `PGBITS`.). For the less function, compare the page addresses.
- **Page for a user address.** Many of the situations where you need to allocate a page have some user address that needs to be accommodated, that is, you need the page containing this address (e.g. the first page for the stack with address `PHYS_BASE - PGSIZE`, the page for a stack access beyond the first page with address > `PHYS_BASE - PGSIZE` or `esp - PGSIZE`, etc.). The page required might or might not be yet allocated. *Hint:* Create a function that returns the page that corresponds to the address in the argument. Allocate, if necessary. Be mindful if the page should be writable or read-only. Furthermore, all page manager functions can be written with an address argument. This will result in a consistent API.
- **Allocation/deallocation.** Allocation of a page means allocating a new `struct page` for the given address with `malloc`. The page should be specified writeable or read-only. (*Hint:* `private = !read_only`.) The structure should be initialized (Hint: No frame,

no sector, no file.) and inserted into the `pages` hashtable of the process. If the insert fails, there is a page like this already, so `free` the allocated structure. Deallocation is the reverse process, but the page might be in a frame (see Paging in/out). In this case, the page should be locked (see Locking), should be paged out, and its frame freed, before deleting from the hash table and freeing the structure. **Note:** There is a condition on paging out this page if it is part of a memory-mapped file that others may still be using See Memory mapped files.

- **Page faults.** In P3, a page fault has a different meaning from P2: instead of signifying a but, it signifies that the page requested has not yet been mapped (i.e. the address requested is outside the process virtual address space currently mapped to pages). So, `page_fault` should invoke the necessary function of the page allocator/manager.
- **Paging in/out** (aka **Loading/unloading**, **Faulting in/evicting**). Essentially, this is the process of writing (the contents of) a page into a physical frame, and, conversely, taking it out. The multitude of names just shows the variety of use cases for this operation. (**Note:** Just like we don't zero out used up stack frames, we don't have to zero out frames after paging out their pages.) The core of paging in, i.e. the actual read-write or copy operation, has to allocate a frame (see Frames for what "allocate" means for a frame exactly) for the page, lock the frame, and then, if swapped out, swap it in (see Swap partition), or if part of a memory-mapped file, read it in from the file (see Memory mapped files), or if a new blank page, zero it out. (*Hint:* This core operation can be encapsulated in a routine taking a (pointer to a) page structure as its argument.) The *paging in* function, taking an address as an argument (**Note:** This will be the function one will call from the new `page_fault` if `user` and `not_present` are both true.) should do the necessary sanity checks (e.g. have hashtable, have page for address), lock the page's frame, if it has one, or call the core routine, if not, and unlock the frame. (**Note:** This is the function that replaces `install_page` so, upon success, it has to register this page with the process page directory, with `pagedir_set_page ()`. This call can also fail, so it needs to be made before unlocking the frame so the boolean return value is correct.) The *paging out* function is the opposite of the *paging in* function, and handles the various use cases for the page. It takes a page with a locked frame, performing the sanity checks on these conditions, then removes the page from the process page directory with `pagedir_clear_page`. (**Note:** This is important to happen first before the rest of the actions to avoid race conditions in case this page is part of a memory-mapped file. After the call to `pagedir_clear_page`, accesses to this page will page-fault, which we know how to handle.) The page is then evicted and disassociated from the frame. (**Note:** A page can be swapped into many different frames, not the same one, after each swap out.) **Note:** If the page is part of a memory-mapped file (that is, in Pintos, `private = false`), instead of evicting the page, its contents have to be written back to the file (called lazy write-back). See Memory mapped files.
- **Lazy loading.** This means that you don't actually "page in" a page until the first access to it (for reading or writing). You get this through `page_fault`. (**Note:** That's why "faulting in" is synonymous to "paging in".) This is true for all pages: pages for executable

and data segments, stack pages, etc. For pages that have to read from a file (e.g. executable or memory-mapped file), you need to set the `p->file` during page_allocate. This will tell you you need to read in from a file and which one. **Note:** For memory-mapped files, all the pages necessary to contain the file's contents are allocated at the same time. See the documentation and requirements for `mmap`.

- **Locking.** The page manager needs to lock a page's frame before manipulating the page, so it uses `frame_lock` and `frame_unlock` (see Frames). Again, it makes sense to encapsulate all the sanity checks into page_lock and page_unlock. *Hint:* `page_lock` can be used in many places as an elegant shortcut, if it not only locks the frame but also checks if the page has a frame at all and pages it in and registers it with the page table, if not. For this you will need a second argument for_writing along with the usual address argument. Example use cases are copying from user to kernel and the `read` and `write` system calls. Don't forget to unlock when done or upon intermediate failure.
- **Copying from user to kernel.** Several different approaches have been taken for copying data from user (process) space to kernel space. Copying a number of consecutive bytes starting at a user address to a kernel address can be encapsulated. Copying a file name or another string (which comes as a pointer to a user-space address) can also be encapsulated. You can get a new kernel page to write it into, making sure that you `palloc_free_page` when you are done with it. **Note:** In both cases, make sure to lock the page for the user address.
- **Recently accessed.** For the eviction algorithm (see Frames), you will need to check if the page has been used recently. The flag is in the page table entry (PTE) and is checked through `pagedir_is_accessed`. **Note:** For the second-chance algorithm to work properly, you need to clear the flag if it was raised at every check. The flag is under software control through `pagedir_set_accessed`.
- **Destruction.** The only case for destruction is `process_exit` in the `exit` system call. The hash table `pages` needs to be destroyed. `hash_destroy` takes a hash table and a callback for properly cleaning up each hash table entry, in this case `struct page`. The callback should free the frame, if any, locking and unlocking it on both ends, before freeing the page structure.

Swap partition

Overview

The swap partition is a (simulated) secondary storage area (e.g. disk) where pages that are evicted from their frames but might need to be paged in again are stored temporarily. We use a block device for our swap partition, which means that it does allocations in blocks. Each block has the same number of sectors, where a sector is the least unit of operation for the block device. Blocks can have different types for different usages, one of which is for a swap partition (`BLOCK_SWAP`).

Both blocks and pages contain multiple sectors. You have to calculate what is the size of a page slot in sectors. While a bitmap will hold a bit for each slot, indicating whether it is empty (0) of full (1), writing pages to the block device (on swap out) and reading them back out (on swap in) will be in the number of sectors that cover a page.

Files

`vm/swap.h`

- Forward declaration of `struct page`
- Swap partition management function declarations

`vm/swap.c`

- The global swap device which is (a pointer to) a `struct block` (see `devices/block.h` and `devices/block.c`)
- A global bitmap to keep track of open slots which is (a pointer to) a `struct bitmap` (see `lib/kernel/bitmap.h` and `lib/kernel/bitmap.c`)
- A global lock to protect the swap bitmap
- `PAGE_SECTORS`, the number of sectors per page (*Hint:* `PGSIZE` and `BLOCK_SECTOR_SIZE` will be helpful)
- Swap partition management function definitions

Data structures

None.

Lifecycle

- **Initialization.** A swap device is created/initialized with `block_get_role (BLOCK_SWAP)`. The bitmap has to be created with the proper number of slots (one page per slot). The lock also needs to be initialized.
- **Swapping out.** Swapping out is evicting a page from its frame and saving it on the swap partition. It is described first, because a page cannot be swapped in if it wasn't swapped out first. Just like with any operation involving a frame, swapping out requires the frame to be locked first. Do the necessary sanity checks. Set the corresponding bit in the bitmap to 1, making sure to synchronize the operation if it is not atomic. The set the page's sector index (the index of the first page slot sector) to the appropriate sector index and `block_write` the page to the sectors of its slot. (**Note:** At this point, we know that this page is not a memory-mapped file (otherwise we would have lazily written back to the file rather than swapping), so the memory-mapping fields of page can be set appropriately. *Hint:* `private` and mapping-related.) **Note:** Outside of this call, make sure you completely disassociate the frame from the swapped out page.
- **Swapping in.** Only a page that was swapped out can be swapped (back) in. Do the necessary sanity checks, including frame locking. (*Hint:* You have to have gotten a frame for the page before making this call.) `block_read` the page into the frame and flip the corresponding slot bit in the bitmap to 0 (emptying it). If the latter operation is not atomic, synchronize it with the lock. Set the page's `sector` appropriately.

Memory mapped files

Overview

Memory-mapped files are a mechanism for reducing the latency of file I/O operations by bringing files into main memory. Since a process interacts with main memory through the mapping of page-sized regions (that is, pages) of its address space to physical memory frames, memory-mapped files are written to frames mapped to process pages, at the time these pages are paged in (lazy load). Any such pages that have been modified by write operations have to be written back to the corresponding regions of the files on disk when these pages are evicted (lazy write-back).

Files

No new files.

`src/userprog/syscall.c`

- `struct mapping` (**Note:** You don't need to use it outside this file.)
- `mmap ()` and `munmap ()` system calls

Data structures

```
/* A kernel thread or user process. Note: Partial. */
struct thread
  {
    /* Note: Other members here. */

    struct list mappings;               /* Memory-mapped files. */

    /* Note: Other members here. */
  };

/* Binds a mapping id to a region of memory and a file. */
struct mapping
  {
    struct list_elem elem;      /* List element. */
    int handle;                 /* Mapping id. */
    struct file *file;          /* File. */
    uint8_t *base;              /* Start of memory mapping. */
    size_t page_cnt;            /* Number of pages mapped. */
  };
```

Lifecycle

- **`mmap` system call.** The `mmap` system call only creates a mapping (allocated with `malloc`) between a file and as many virtual process pages as would be necessary to contain the whole file. There is no reading or writing. You need to allocate all the pages that would "cover" the file, initialize them properly, and add them to the `pages`

hashtable. As usual, on any failure, you should clean up and report the failure with the proper error code. Since, per the `mmap` examples, the file will already be open, use the function `file_reopen` instead of `file_open` to get a file pointer to assign to the mapping. *Hint:* Be sure to read the requirements in the Manual carefully, comply with them, and use locking where appropriate. **Note:** The Pintos `mmap` system call does not have a `private` argument like Unix/Linux. All file mappings are therefore public. Set the private member of the pages accordingly.

- **Mapping id.** You can reuse your file handle generator for the mapping id. All files and mappings will have distinct id-s within a process. Depending on the system call, you have to look up the handle argument either among your file descriptors or your file mappings.
- **`munmap` system call.** The munmap system call is the opposite of mmap, in that you will remove the pages from the hashtable, deallocate them, and close the file that was open for the mapping, and `free` the mapping structure.
- **Dirty pages.** On page out for a memory-mapped file page, per the requirements in the Manual that the file itself is used as a backing store for the mapping, the page has to be written back to the corresponding region in the file *if and only if* it has been modified. There is a hardware-set flag in the page's page-table entry that is used to check for this condition, and the corresponding function is `pagedir_is_dirty`.
- **File data consistency.** Since in Pintos all file mappings are non-private, we are not required to maintain data consistency across different mappings of the same file. This means that if two or more processes map to the same file on disk, each process may see file content changes that are inconsistent with the operations it has performed on the file.

Second-chance algorithm

Overview

The second-chance algorithm is a clock-like eviction-policy algorithm. It is one way to implement the choice for page eviction in the case of running out of free frames. The algorithm is "clock-like" because it scans the frame array in a circular (modulo) manner. The scan pointer is called a "hand". The algorithm is "second-chance" because it does not evict pages that have been recently accessed.

Details

- **Frame availability.** A frame is available if it is not associated with a page, and, conversely, it is unavailable when it is associated with a page. The struct frame and struct page hold mutual pointers. *Hint:* Always initialize and clear your pointers to `NULL` when they are not supposed to point anywhere. This allows you to do sanity checks and assert expected prerequisites for page management.
- **Hand movement.** The hand is essentially an index into a large array (the frame array). The hand "moves" by being incremented modulo the array size, giving it "circular

motion”. When no frames are available, an eviction scan begins, starting at the current position of the hand and going around for two full revolutions. Pages are checked for eviction as the handle points to them. **Note:** The two revolutions allow for giving a second chance to recently accessed pages.

- **Recently accessed pages.** Each page to which the handle points is checked for recent access. The check is based on the `pagedir_is_accessed` function, which examines a flag set by hardware in the page table entry for this page. If the page has been accessed, it will not be evicted this time around, but its access flag is cleared using the `pagedir_set_accessed` function. *Hint:* This functionality should be encapsulated into a page manager function.

Sketch

Frame index	Page pointer	Page accessed
56	non-`NULL`	0
57	non-`NULL`	0
⇒ 58	non-`NULL`	1
59	non-`NULL`	0

Note: The hand is currently at 58. The page accessed flag will be cleared. If the page at 59 does not get accessed before the hand is advanced to it, that page will be evicted.

Workshop 6

Prereqs

OSPP: 2, 3, 4, 5, 7.1-5, **8.1-3**, **9**, 11, 12
Pintos: 3, **4**, **A.4.2-A.8.7**
Files: All files mentioned in the designated manual reading, including the programs used for the tests.

Tasks

Note: *Teams who are still working on P2, you need to finish P2 and P3-prep by Sun, May 7. There will be office hours available every day 12:00-18:00 Tue-Sun.*

UCD	MSUD	Task

cardinal	red	P3 - Paging: problem, files, tests
violet	*orange*	P3 - Stack Growth: problem, files, tests
black	yellow	P3 - Memory Mapped Files: problem, files, tests
amber	*green*	Page directories and tables, address translation
gray		Block devices and the swap partition
white		Frames and page eviction policies