MSK
AT
nCA
PL
YH

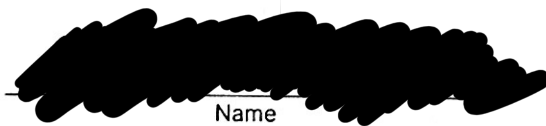
CS 111: Operating System Principles

Midterm Exam Fall '21

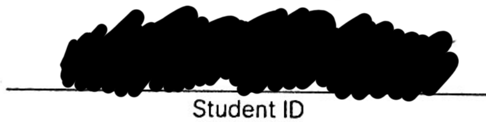
Instructor: Jon Eyolfson

October 26, 2021

Duration: 1 hour 50 minutes



Name



Student ID

This is a closed book exam. You are only permitted a pencil or pen.
Answer the questions directly on the exam.

If in doubt, write your assumptions and answer the question as best you can.
There are 8 numbered pages (page 8 is blank if you need extra room).
The pace of the midterm is approximately one point a minute.
There are 100 total points.

Good luck!

(5 points) Explain the concept of virtualization and how it applies to operating systems.

Virtualization tricks each process into thinking that it has access to all the resources on your computer - it is a useful abstraction. For instance, each process has virtual memory, which means that each process thinks that it has access to all the memory.

5

— **(5 points)** What service would you find in a monolithic kernel, but not in a microkernel?

Monolithic kernel does most things in kernel mode, microkernel runs minimal number of things in kernel mode. File management is done in a monolithic kernel, and IPC.

5

— **(5 points)** What should you use to monitor all system calls a process makes?

You can use strace - this records all the system calls that you make.

5

— **(5 points)** If you include a C `struct` in your library's header, why shouldn't you ever change it? Why?

If you include a struct in your library's header, that means you are exposing it. Now if you change the ABI, for instance by re-ordering the fields, you would break any program that indexed into your struct, either directly or under the hood. The offsets will be different if you reorder the fields.

5

(5 points) What are the two responsibilities of `pid 1` (`init`)?

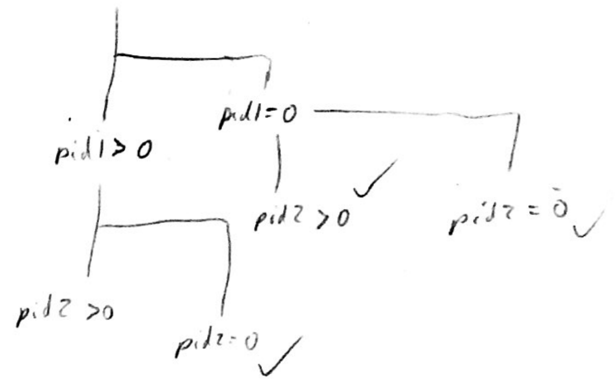
It is the first process that starts up, and is responsible for spawning all the other processes as children. It also serves as the default "orphanage", which means that orphaned child processes will get re-parented to init, and init will call wait and acknowledge them.

5

(5 points) Why do we not use the least recently used algorithm to do page replacement in practice?

Due to hardware and software constraints, it wouldn't be practical. If you implemented using hardware, you would have to iterate through all the pages to find the least recently used, which is inefficient. If you implemented with software using a doubly linked list, you would have to reorganize a lot of pointers and it is still inefficient.

5



(20 points total) Process API.

Consider the following code:

```
#include <sys/wait.h>
#include <unistd.h>

int main() {
  pid_t pid1 = fork();
  pid_t pid2 = fork();

  if (pid1 > 0) {
    int wstatus;
    wait(&wstatus);
  }

  if (pid2 > 0) {
    int wstatus;
    wait(&wstatus);
  }

  return 0;
}
```

called twice

called twice

wait is called 4 times, only need it 3 times

Recall that `fork` creates a new process, that is a copy of the current running process. It returns a process ID, `pid`. If `pid` is greater than `0` then it represents the process ID of its new child process. If `pid` is equal to `0` then this process is the new child process. We'll assume these are the only possibilities, `fork` never generates errors. The `wait` function waits until one of its child processes terminates, and reads its status information so the kernel can remove its resources. We can assume that all processes exit normally. We don't need to access the information in `wstatus`, so for this question it's irrelevant. We also don't check the return value, so we don't need to know it for this question.

We compile the program on the previous page, and execute it as a new process, pid 100. Again, we assume that fork does not fail, and all processes that terminate exit normally.

2

(2 points) How many *new* processes get created (exclude pid 100)?

3

8

(8 points) Does pid 100, or any of its children create any orphan processes? Why?

No, all of the children are acknowledged. See diagram on previous page. The "if (pid2 > 0)" statement ensures that both children created from the second fork are acknowledged by its parent, since this code will be executed twice. The "if (pid1 > 0)" statement ensures that the child created in the first fork is also acknowledged, so all children are acknowledged and there are no orphans.

10.

(10 points) There's an issue with this program. When you run it, it seems fine, but that's because we don't check for any errors. What is this issue, and how would you fix it? (You can just describe what you'd need to do to fix it, instead of writing code.)

Wait is called more times than needed
It is called 4 times, even though we only need to wait on 3 children. ✓ You get an error if your program calls wait too many times, since it will try and acknowledge a child that doesn't exist.

You could fix this with an added condition in the first if statement, by changing it to
if (pid1 > 0 && pid2 > 0) instead of
if (pid1 > 0)

This ensures that we only call the wait in this if statement once, to clean up the one child that we created in the first fork.

(20 points total) Basic IPC.

Consider the following code:

```
#include <sys/wait.h>
#include <unistd.h>

int main() {
  int fd[2];
  pipe(fd);

  pid_t pid = fork();
  if (pid == 0) {
    /* first child */
    dup2(fd[1], 1);
    close(fd[1]);
    execlp("ls", "ls", NULL);
  }
                  ← parent still has access to open write end

  pid_t pid = fork();
  if (pid == 0) {
    /* second child */
    dup2(fd[0], 0);
    close(fd[0]);              ← need to close(fd[1])
    execlp("wc", "wc", NULL);
  }
  else {
    close(fd[0]);
    close(fd[1]);
    int wstatus;
    wait(&wstatus);
    wait(&wstatus);
  }

  return 0;
}
```

Recall that pipe creates two file descriptors: `fd[0]` and `fd[1]`. You can only write data to `fd[1]` and only read data from `fd[0]`. The `close` function takes a file descriptor as an argument and closes it, allowing the kernel to clean up the entry. The dup2 function copies the file descriptor in the first argument to the file descriptor in the second argument. If the file descriptor represented by the second argument already exists, it's closed before the copy. The `execlp` function takes a string, representing an executable name, and any number of string arguments terminated with a null pointer. The function searches for the executable and if found, replaces the currently running process with that one (also passing the arguments provided). Assume that no functions ever fail.

We need to use read and write system calls because the pipe is stored in the kernel, so we need to use the system call interface.

We compile the program on the previous page, and execute it as a new process. Again, we assume `fork`, `pipe`, `dup2`, and `close` do not fail, and all processes that terminate exit normally.

(10 points) Explain how `ls` and `wc` communicate using the pipe. You should explain it in terms of each process, and the `read` and `write` system calls (both functions operate on a file descriptor and a sequence of bytes). The pipe has a read end and a write end. It is essentially an internal buffer that the kernel creates. ls takes input from stdin, and we used dup2 to redirect the output to the write end of the pipe. The write end is a file descriptor, so ls makes a write system call with the write end of the pipe as its file descriptor. Once the information is in the pipe and we closed the write end, we can read from the read end. We redirected the input to wc to the read end of the pipe, by changing wc's file descriptor. Wc makes a read system call with the new file descriptor, and the process can communicate using the pipe.

10

(10 points) When you run this program, it looks like it hangs. Why? What would you have to do to fix it?

It looks like it hangs because the program is still waiting for input, since we haven't closed all the write ends of the pipe. If we don't, then the terminal assumes that the pipe should still be waiting for input, and so it hangs. We need to close all the write ends so that we get the sign that we have reached the end of the file, and we can start reading from the read end of the pipe.

10

When we call the second fork, that child does not close the write end of the pipe. It inherited an open write end from the parent, and so it needs to close it. We can fix the issue by calling close(fd[1]) in the second child.

(20 points total) Scheduling.

Consider the following processes you'd like to schedule:

Process	Priority	Arrival Time	Burst Time	response	waiting
P_1	2	0	4	0	0
P_2	1	5	3	2	5
P_3	1	7	2	2	2
P_4	1	1	3	4	8
P_5	2	3	1	1	1

You decide to use a round robin scheduler with a quantum length of 2 time units, and a priority queue. You decide that a larger priority number means a process has a higher priority. You decide to schedule the processes such that a higher priority process always runs ahead of a lower priority one. Processes that have the same priority round robin normally.

(13 points) Fill in the boxes with the current running process for each time unit.

13/

arrival times: P1 (0), P4, P5, P2, P3

0 ... 13

P1	P1	P1	P1	P5	P4	P4	P2	P2	P3	P3	P4	P2

(3 points) What's the average response time? (Your answer can be fractional.)

3/

9/5 time units

(4 points) What's the average waiting time? (Your answer can be fractional.)

4/

16/5 time units

(10 points total) Page Tables.

Your system has 2 MiB (2^{21}) pages, a PTE size of 8 bytes, and uses 57 bit virtual addresses. You decide to use multi-level page tables, and fit each smaller page table on a single page.

(2 points) How many PTEs can you fit into a single smaller page table? (Answer can be a power of 2.)

2

PTE = 2^3

Page = 2^{21}

2^{18}

(4 points) How many levels do you need for your multi-level page table? Show your work.

4

21 offset bits, since page size = 2^{21}

57 − 21 = 36

Index bits = 18

$\left\lceil \frac{\text{virtual - offset}}{\text{index}} \right\rceil = \left\lceil \frac{36}{18} \right\rceil = 2$

Now that you have a multi-level page table with *n* levels (if you didn't calculate *n* you can assume a value greater than 1). You want to calculate the effective access time of this approach. On this system it takes 10 ns to search the TLB, each memory access takes 100 ns, and we have a hit rate of 50%. Recall that for a single page table the equation for effective access time is:

$\alpha = 0.5$

$$\text{EAT} = \alpha \times \text{TLB}_{\text{HitTime}} + (1 - \alpha) \times \text{TLB}_{\text{MissTime}}$$

where

$$\text{TLB}_{\text{HitTime}} = \text{TLB}_{\text{Search}} + \text{Mem}$$

$$\text{TLB}_{\text{HitTime}} = \text{TLB}_{\text{Search}} + 2 \times \text{Mem}$$

(HitTime crossed out, "misstime" written below)

multi not single

(4 points) Calculate the effective access time for this multi-level page table. Show your work.

2

$\text{TLB}_{\text{HitTime}} = 10 + 100 = 110$

$\text{TLB}_{\text{MissTime}} = 10 + 3 \times 100 = 310$

If we miss, we need 3x Mem because we have 2 levels of page tables, need to go into memory once for each page table and once more for actual value

EAT = $0.5 \times 110 + 0.5 \times 310$

= 55 + 155

= **210 ns**