Midterm questions:

BOOTSTRAPPING

void wait\_for\_disk() {

while (inb(0x1f7) != 0x40) //polling status registor 0x1f7: two directional register

continue;

}

void read\_ide\_sector(int s, char\* a) {

wait\_for\_disk();

for(int i = 0; i < 4; i++)

outb(0x1f3+i; s >> 8\*i);

outb(0x1f7, 0x20);

}

1a. omit “outb(0x1f7, 0x20);”. What will go wrong?

If we do not copy 0x20 to 0x1f7, which is a status register, the program will not know if the sector reading has finished or not. It might enter an infinite loop, waiting for the status to be ready.

1b. replace uint16\_t with char. What will go wrong?

uint16\_t is a two-byte unsigned integer, where char only holds one byte. When the input is over 8 bytes, the behavior is wrong. The data might get truncated or messed up.

2a. Watershed effect applies to bootstrapping?

We could have used a smaller buffer other than char[512]. Though it might save some space, the program now can only count words less than 512 characters. The *utilization* of the program is reduced.

2b. Economy of scale in scheduling

When many threads come in Round Robin scheduling, the average waiting time is short compared to FIFS and SJF because every job will have a chance to start after it arrives.

3. Double buffering helps performance in some cases. What about triple or quadruple buffer?

Triple and quadruple buffering can also help performance in some cases. When one buffer is reading, and the other processing, the others could serve as a drawback. If a buffer goes wrong, the third buffer can correct it.

Triple and quadruple increase latency. The others do not get processed until the buffers in the front are done.

4. Why critical sections are needed in a single-threaded environment?

fd = open(“abc”, O\_RDWR);

foo(fd);

write(buffer, fd);

close(fd);

5a. The program will execvp in the parent process.

5b. The program does not detect where the child program has exited normally with an exit code.

5c. The switch statement will fall-through. When fork() fails or fork() to a child process, it still execvp and causes failure.

5d. When fork() fails, it execvp. When fork() enters a child process, it returns -1;

5e. \_exit() does not flush buffer when child dies, which will not mess up the parent process’s stdout. exit() does.

5f. pid\_t p = fork(); fork();

The parent process has two children running in parallel. Each one is a copy of itself. The switch statement only cares about the first child. The action of the first child will duplicate.

5g. The program will execute cat will NULL, which might fail if there is no specified stdin or stdout elsewhere.

Instead of issuing a CALL FOO instruction, it pushes FOO onto the stack, and then executes an INT 92, which pops FOO from the stack and then jumps to FOO.

6a. This idea will work as long as INT 92 will direct the instruction pointer from the user mode to the kernel. The kernel of the Linux machine must have the ability to save the return address so that the insn pointer can return to the user mode after performing a syscall.

6b. It will not attain hard modularity if FOO fails, and the instruction pointer jumps to FOO, it will not be guaranteed that it can return to where it was. The propagation of effect still exists. Without CALL FOO instruction, it loses a boundary that can catch FOO’s failure.

7. Round-robin policy, whenever a quantum expires, we choose a process at random. RR is fairer.

A 1,5

B 2,2

C 3,3

D 4,3

RR: ABCDABCDACDACA

RRR:ABCDDCCDDBAAAA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | RRR-response | RRR-turnaround | RR-response | RR-turnaround |
| A | 0 | 16+7a | 0 | 16+15a |
| B | a | 9+6a | A | 5 |
| C | 2a | 5+4a | 2a | 13 |
| D | 3a | 6+5a | 3a | 8 |

RRR has a lower average turnaround time, and it does not perform as many context switches. They have the same wait time.

8. What is a process can gain a lock that it already has?

A deadlock might occur. If a process holds a lock and can hold it again, when it is unlocked, one of the locks is unlocked but the other one still exists. The other thread will not enter this section.

If a process can unlock twice, it might unlock a lock for something else.

1. Race condition of shell script code:

echo a >> out | echo b >> out

out is uncertain

2. Diseconomy of scale when critical sections are used.

If there are 10000 threads trying to run through a critical section that cost each one time t. Now it would cost 10000t for all.

3. while(inb... || !inb...)

calling inb() twice is bad. If the byte switches between each call, the loop could go wrong.

4. Time-space trade-off in the implementation of a scheduling algorithm.

If we have a priority table that stores the order that we should run processes, optimize time but cost more space.

5. Maximum throughput in FIFO, SFJ, LIFO, LJF? Maximize turnaround time?

SJF has the maximum throughput because most jobs get done and not waste on context switches. LJF has the maximum turnaround time because all other shorter jobs are pushed behind the longest job.

6. Is the choice of scheduling algorithm orthogonal to whether hard or soft modularity is used?

No. In RR, often context switches require harder modularity so that each process does not affect each other.

7. Use JMP to make system calls?

It will work because as long as the JMP instruction can jump from user to kernel mode. The kernel can look up the location of the actual system call, store the destination of JMP as return address, and direct the instruction pointer to the actual system call.

8. Execution order for each system call by all of the processes? Each file contains “OUCH\n”

true & false fork, execvp true, execvp false, waitpid

sleep 60 | sleep 1 fork, pipe, dup2, close, execvp sleep 60, open, dup2, close, execvp sleep 1, close, waitpid

cat foo | echo foo open, pipe, fork, dup2, close, execvp cat, open, dup2, close, execvp, close, waitpid

(rm foo **&&** cat) < foo open, fork, dup2, close, execvp rm, unlink, open, dup2, close, execvp cat, waitpid

execute cat only if rm foo succeeds

9a. not: takes a command and its arguments, executes the command, and then succeeds iff the command fails.

waitpid

if exitstatus != 0

return true

9b. not not ... work?

No. If there is an error arise in the first not, it will fail, but the not... actually fails and not not should return true. We cannot determine if it fails because not... succeeds or not has a problem.

10a. balance is negative.

10b. critical section from b1 = ... to amount += ... OR critical section all.

10c. cmpxchg(balance, b1, b+amount);

1. Does Ubuntu use soft or hard modularity?

Ubuntu uses soft modularity. Ubuntu is a linux distribution that uses linux kernel, which is a monolithic kernel. A monolithic kernel adopts soft modularity because all parts of the kernel share a common address space. This reduces context switches because are buggy.

2. echo four | lab0 –output=score –output=and –output=7 --output=years --output=ago

ago contains four. All other files are empty.

3. INT instruction to execute system call. Is the idea valid?

Yes, it is valid. As long as the interrupt provides a way to trap from user mode into kernel and returns to the user mode, the system call execution is valid.

4a. read() on pipe will only return 0 if all write ends to the pipe are closed, otherwise, read() will just block there forever.

**$ (head −n 20 2>a <b | sort 2>>c | tail) >d**

./simpsh \

--creat --wronly a \

--rdonly b \

--append --wronly c \

--creat --wronly d \

--pipe \

--pipe \

--command 1 5 0 head -n 20 \

--command 4 7 2 sort \

--command 6 3 3 tail \

--close 5 \

--close 7 \

--wait

**$ cat <d | cat >>d**

./simpsh \

--rdonly d \

--creat --append --wronly d \

--pipe \

--command 0 3 3 cat \

--command 2 1 1 cat \

--close 3 \

--wait

4b. The original head will treat -n 20 as the standard input of head. Since 2>a is ahead of <b, head will output to a before it gets the standard input. The simpsh will work as head -n 20 2>a <b.

4c. cat < d | cat >> d

reading and writing at the same time will cause the shell waiting forever. The writing end will never be closed.

4d. Upward-compatible changes to simpsh that will allow you to translate the script to simpsh faithfully.

Consider the order of opening files. For example, if a command has not redirected any standard input before redirecting stdout or stderr, the command either takes its argument as stdin, or does not have a stdin. The command will not dup2 stdin after stderr or stdout.

5b. T2R: Round two robin, prob(0.5) that the current process will continue running when it times up. Starvation is possible with T2R because there is a chance (0.5)^∞ that a process will never be moved down. Even though the number might get really small, it is still not an absolute zero. The process that is at the bottom of the queue will get starved.

6.

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

static unsigned char n;

void handle\_sig(int sig) {

printf("Got signal! n=%d\n", n++);

}

int main() {

signal (SIGINT, handle\_sig);

do {

printf("looping n=%d\n", n++);

signal (SIGINT, handle\_sig);

}while (n!=0);

return 0;

}

6a. output more than 256 lines

a signal comes in when n=255 and incremented to 0, and n becomes 1 in the signal handler.

6b. print two n=N

a signal comes in between two print statements

6c. output two ==

a signal comes in at the printf function. The behavior is undefined.

6d. dump core

signal in the printf function: dump core

printf function is not thread safe.

6e: remove which line does not have an effect

return 0;

The second signal (SIGINT, handle\_sig);

removing line 9: if a signal comes in before do-while loop, it will not be handled.

7.

void wait\_for\_ready() {

while ((inb(0x1f7) & 0xC0) != 0x40)

continue;

}

void read\_sector(int s, char\* a) {

wait\_for\_ready();

outb(0x1f2, 1);

outb(0x1f3, s & 0xff);

outb(0x1f4, (s>>8) & 0xff);

outb(0x1f5, (s>>16) & 0xff);

outb(0x1f6, (s>>24) & 0xff);

outb(0x1f7, 0x20);

wait\_for\_ready();

insl(0x1f0, a, 128);

}

7a. Remove second wait\_for\_ready();

read trash data left.

7b. outb(0x1f3, s & 0xfff);

nothing changes. only sig byte.

7c. Interchange outb(0x1f3, s & 0xff); and outb(0x1f4, (s>>8) & 0xff);

nothing changes.

7d. Interchange outb(0x1f6, (s>>24) & 0xff); and outb(0x1f7, 0x20);

wrong data. We change status register before finishing reading.

7e. Put another copy of wait\_for\_ready(); before insl(0x1f0, a, 128);

increase runnning time because it will wait the disk twice.

8.

#include <unistd.h>

int main (void) { return fork () < fork (); }

create four child processes.

sequence: \_exit(), \_exit(), \_exit(), \_exit()

1. fork() - child, child

2. fork() - child, child