Task 2 "memsize"

Questions — not for submission

1. How does calling a system call differ from calling a regular function? How does it work?
2. How are parameters passed to the system call function? How is the return value returned to userspace?
3. What is the purpose of the `usys.pl` file?
4. What is `struct proc` and where is it defined? Why do we need it? Does a real-world operating system have a similar structure?
5. How much memory does our program use before and after the allocation?
6. What is the difference between the memory size before and after the release?
7. Try to explain the difference before and after release. What could cause this difference? (Advanced: look at the implementation of `malloc` and `free`).

1) סרטן פונקציה - ה-PC קופץ למקום אחר בקוד, אותו עדיין ב-userspace

קריאה לפונקציה מערכת - אותו עובדים לקרנל מוד - מקבלים יותר הרשאות ונגדר הגבלים בגרעין שאר

ב-trapframe (רגיסטרים וכו').

כך בעזרת:

user code (like memsize()) defined in user.h and exec' by usys.pl

→

prepare systemcall (get register to use for sys call "a7")

→

call ecall (switch to kernel mode)

it save user program state

→

in kernel mode, checks if trap caused by system call

→

call system call

→

use trapframe to access syscall info (like read args)

→

return value in trapframe

→

return to usermode and restore state

→ return the method data

2) הפרמטרים מעוברים לפונק' system call ברגיסטרים a0-a5 ב trapframe

הקרנל קורא אותם בעזרת פונק' כמו argint() argint(),

הערך המוחזר נשמר ב a0 בתוך trapframe כאשר הקרנל חוזר ל usermode.

3) לקיצוץ טקסט שמבצע קוד אוסף, שימוש בין הקריאה מדר משתמש לגרעין בפעולה בעזרת -

בחיבור a7 את שם הקריאה הנערכת, שליחת interrupt (ecall) (כמו 0x80 שמעבר לקרנל)

ואיגוד כפול דאג מהמערכת מגבה למשתמש.

4) כאן הנדנה עמל הגרעין שמעורבת בדיבה על הבלוקים (PCB) שבגרירה ל proc.h

רק נגע אות מדר הבלוקים. מגרעין על בגיבוי שלו וכו'. (mypproc()).

בלינוקס הסיקרון דומה אבל שלו בשרש יותר.

5) הגדיל כאוצר ה - מסקסט bytes. (עשה יותר מ-20 כי איבוד / malloc בגירה יותר קל)

6) לא השנה ה sizeof לפני ואחרי השחרור.

(לא השנה כי הוא רק מגדיר טוח ב memory reuseable)

7) (עשה יותר מ-20 כי איבוד / malloc בגירה יותר קל)

(לא השנה כי הוא רק מגדיר טוח ב memory reuseable)

Task 3

5. Modify the shell to print the exit message when a child process of the shell terminates.

For submission: `goodbye.c`, modified `sh.c` and any modified OS files.

Questions — not for submission

1. What happened as soon as we changed the signatures for `exit` and `wait`? Why?
2. Why do we need to add a new field to the PCB for the exit message? Why can't the shell just read the message from the exiting process?
3. When does the shell get the exit message from the child process? Does it happen immediately after the child process exits?
4. What happens if the exit message is *longer* than 32 characters? How do we make sure nothing bad happens?
5. What happens if the exit message is *shorter* than 32 characters? How do we make sure nothing bad happens?
6. How many times is our exit message copied?
7. Where in `sh.c` does the shell receive the exit message? Explain briefly how this code works.
8. What happens if the shell modifies the exit message after it is received?

1) user-side ו-kernel-side לא מאוכלסים אותו אותם בשני ניגרר אוטומטית כי כל אחד מוגדר בנפרד שונים. צריך לזכור בהקשרים שונים כמו syscall table/user.h עבור syscall, וכו'.

2) צריך להוסיף field חדש ל PCB, exit_msg כי כאשר קורה exit() הילד נכנס משותר. ה shell לא יוכל לקרוא לו מבלי נכון. אותנו מעניינת את הבורא ל exit_msg לפני שחרור הילד אותנו שומרים אותה בזיכרון הקרנל כשהילד שבנוכב יקרא בפעם הבאה.

3) לא, exit → kernel save msg in p->exit_msg →

child become zombie -> still in process table waiting for parent to clean up

→ the shell calls wait(&status, msg-buffer)

→ copy the msg using copyout

→ only during wait gets the message

4) משתמשים ב safestrcpy() ב exit שיעתיק ל exit_msg.

יכניס רק עד הגודל שהגדרנו אם יותר ממנו, בלי שזו יהיה בעיה.

5) גם בסדר שגם קצר יותר, safestrcpy() מוסיף null-terminated סמל שהקטע הסתיים רק שזו נמשיך כרגיל.

6) בעיות.

- בעיה אחרת לקרוא ב exit שמשתמשים ב wakeup() ואם safestrcpy().

- בעיה אחרת מהקרנל של הילד לבורא buffer space user (wait)

7) ב sh.c בפונקציה ב main, נקבל אור ב exit_msg אחרי שקורה wait.

שומר אור הבורא מהילד PCB לתוך shell buffer. ואז מדפיסים.

8) אותו לוקחים אור הבורא מהילד ומשחררים לו אור בזיכרון, פעם בפעם שעורך אור בילד אותו נערוך רק אור ב buffer שנמצא ב shell.

Task 4

Questions — not for submission

1. How does `fork` work? How does it create a new process?
2. How does `wait` work? How does it wait for a child process to finish?
3. How does `wait` decide which child process to wait for? How does it decide whether to sleep or return immediately?
4. What is the purpose of `wait_lock`? Why does it appear in `exit` and `wait`?
5. What is the difference between our new system calls and the existing `fork` and `wait` system calls?
6. What is the main advantage of using `forkn` and `waitall` over using `fork` and `wait` multiple times?
7. How come the child processes have access to the same memory as the parent process? What happens if a child process modifies the memory? Can a real-world operating system optimize this?
8. Why do the child processes need to return their exit status to the parent process?
9. Why does the parent process need to wait for the child processes to finish?
10. What is the overall lifecycle of a process in xv6? How is it expressed in the code? How is it reflected in functions such as `fork`, `exit`, `kill`, `wait` and `sleep`?
11. What do the internal kernel functions `sleep` and `wakeup` do? How are they used in the kernel? How are they related to the process lifecycle?
12. What makes the sum operation suitable for breaking up and distributing among multiple processes? What are the costs to be paid for this?
13. Experiment with different array sizes and number of child processes. What happens if the array size is too small or too large? What happens if the number of child processes is too small or too large? Since we are running inside an emulator on different machines, your results may vary. You might not observe any differences in performance at all.

1) fork() יוצר תהליך חדש עליו יוצרים PCB חדש, להעתיק את נתוני המרחב לתוך (בלוק) לבוי שלו Runnable.

הוא כמו אורח שלו יוצא עם PID שונה ו return value.

parent ← return value child PID

child ← return 0

בעזרת allocproc() יוצר PCB חדש.

uvmcopy() מעתיק את המרחב בזיכרון

trapframe מעתיק ומעדכן את ערכי הקונטקסט שלו

2) wait() בודק אם ה process table מכיל ילד במצב zombie ה תהליך בשימוש כרגע.

אם קיים מוצא, מחזיר את הילד PID ו exitstatus.

אחרת אם מוצא, משחרר בזכרון במצב sleep עד שילד עושה exit.

משתמש ב wait_lock בשביל לגשת ל process table ו sleep שיחזור לרץ.

3) הראשון שמוכן ה zombie.

אחרת מוכן ← יושן.

אם אין ילדים רלוי מחזיר ל-1.

4) wait_lock שומר על הגנה וגם בזמן wait ו exit.

- בלעדיו כשמישהו exit או מגיע בודק ל zombie ה process table ישאר כמו שהיה לבוי.

למנוע race conditions / 2 תכונות מנסים לשחרר את אותו בלוק / למנוע deadlock.

5) wait/fork מדורדרים על תהליך אחר.

- שלנו עושים כמה בשבילו.

6) קוד יותר קריא, קל יותר, מסודר ומונע דליפות. פחות קריאות שלא משתמשים.

7) אולי גם deepcopy.

ב real-world גם כאן מתבצע בקונפליקטים, שבה שונה בין דרישות שונות לבין עדכון הפרטי לחסום ביניהן.

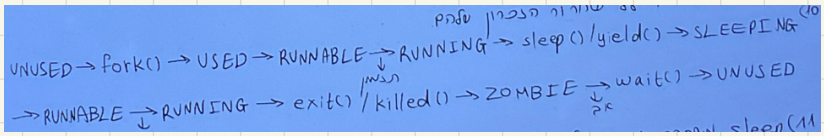
8) שהגורם יגע גם קורא אולי לא יגלה. בגילוי/נרשם

9) - לטפל zombies ולנקות תהליכים ולא.

- waiting מאוטר לקבל טוב ה results של בילד

- בלי לחכות, הזכרון יהיה leaks והבורה מאוד שיטה ע"ז בילד.

10)



11) sleep() יושם בלוקים לתהליך עד שיגשו בסיום קורא

wakeup() מתקבלים בתהליך שמאוזנים קורא

- קוראים ל- sleep כשאנחנו מחכים שמן מלא יסיים (לא ביטלו עדיין) (wait)

- קוראים ל- wakeup שמן מסיים לעדכן אותו. (exit)

12) אופטימלי לא גם במצוקה בתקרוב בתלות תלויים. כמו שסדר בחצרות בתשובה לא משנה.

- יכולים הרבה תהליכים כבר ומתקדם בכדי בכדי ביניהן.

כלל לא מגיעים בדרך משנעות, עכשיו אחד יהיה הרבה יותר קרוב.