OS161 PA2

System Calls

Deliverables

- Answers to the code walk-through questions

- Design Document

- Implementations
  - System calls
    - getpid
    - fork
    - execv
    - waitpid
    - exit
  - Multi-level queue scheduler

Deliverables (contd.)

- Design document

  - A high level description of how you are approaching the problem

  - A detailed description of the implementation (e.g., new structures, why they were created, what they are encapsulating, what problems they solve)

  - A discussion of the pros and cons of your approach

  - Alternatives you considered and why you discarded them

Configure and Build Kernel

- Repeat the steps you used for PA1
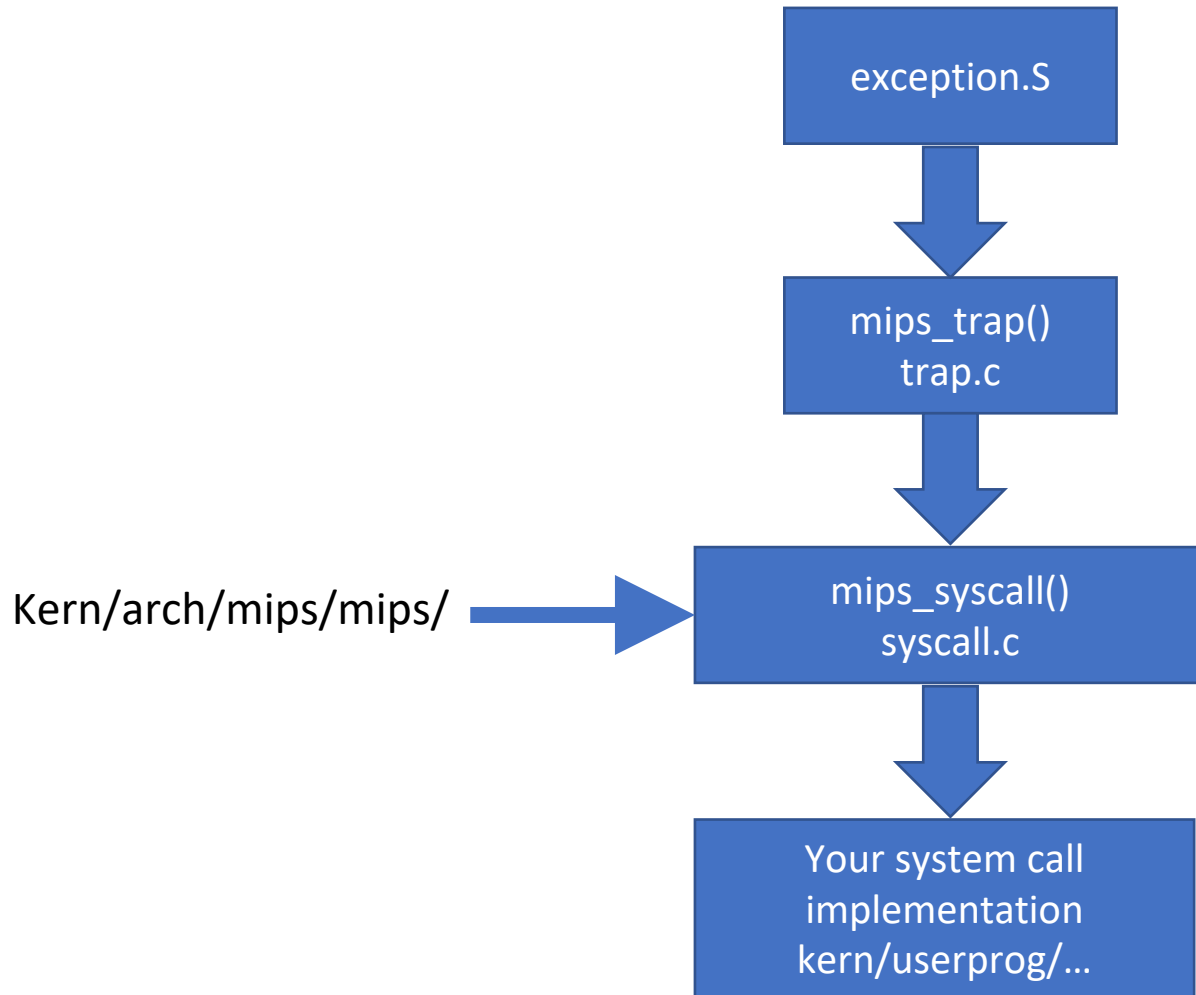 - Just use ASST2 instead of ASST1

Where to put your system call implementation?

- This time no skeleton code is given
- Create under kern/userprog
  - fork.c
  - execv.c
  - waitpid.c
  - getpid.c
  - exit.c
- Name your system calls sys_{getpid|fork|execv|waitpid|exit}
- Add the new files to kern/conf/conf.kern
  - e.g., file userprog/getpid.c
  - The same way you have done hello.c in ASST0.
  - Include your system call function declaration in kern/include/syscall.h

User-Level Interface

- os161-1.11/include/unistd.h contains the user-level system call interfaces.

  - int execv(const char *prog, char *const *args);
  - pid_t fork(void);
  - int waitpid(pid_t pid, int *returncode, int flags);
  - int open(const char *filename, int flags, ...);
  - int read(int filehandle, void *buf, size_t size);
  - int write(int filehandle, const void *buf, size_t size);
  - int close(int filehandle);
  - int reboot(int code);
  - int sync(void);

How is it linked?

```
                    ┌─────────────────┐
                    │   exception.S   │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │   mips_trap()   │
                    │     trap.c      │
                    └─────────────────┘
                             │
                             ▼
Kern/arch/mips/mips/ ──────▶┌─────────────────┐
                    │  mips_syscall() │
                    │    syscall.c    │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Your system call│
                    │ implementation  │
                    │  kern/userprog/…│
                    └─────────────────┘
```

For each system call

- mips_syscall(struct trapframe *tf) gets called

- The system call number is in tf->tf_v0

- The arguments are in tf->tf_a0, …, tf->tf_a3

- Case-switch statement calls the correct system call based on call number, and passes the arguments extracted from the trapframe

✓ Increment user-program counter before returning from system call
  Otherwise, it will restart the same system call
  tf->tf_epc+=4

✓ If error
  Store the error code in tf->tf_v0
  Set tf->tf_a3 to 1

✓ If no error
  Store the return value in tf->tf_v0
  Set tf->tf_a3 to 0

Process structure

- A common hack.
  - Add the necessary fields to the thread structure and treat it as a process.
    - Pid
    - Exit status
    - Parent process
    - Etc.
  - A *process table*
  - A new pid needs to be generated for a new process
    - Need to reuse pid of processes that already exited

Sys_getpid

- Simplest one.
- Just return the pid of the executing process.
- getpid does not fail.

Sys_fork

- Duplicate the current process.
    - Child process will have unique process id.
- pid_t sys_fork(struct trapframe*tf, pid_t*retval)
    - Child process returns 0.
    - Parent process return the pid of the child process.
- In case of an error
    - do not create a new child process but return -1.
- Most of the work is already done in thread.c (thread_fork). Add the followings:
    - Create a pid when creating a new process. Add it to your process table.
    - Copy the trapframe.
    - Copy the address space.
    - Call thread_fork()

Sys_fork

- Implement md_forkentry
  - Parent's trapframe and address space are passed as arguments
  - Create new child trapframe by copying parent's
  - Get the assigned child pid from parent's trapframe tf_v0 and assign it to the pid of the current process (since we are executing md_forkentry, this is child)
  - Set the trapframe's tf_v0 to 0.
  - Increment tf_epc by 4.
  - Copy the passed address space to the current process address space and activate it.
  - Give the control back to the usermode.
    - Call mips_usermode() and pass the new trapframe.

Sys_fork errors

EAGAIN

Too many processes already exist.

ENOMEM

Sufficient virtual memory for the new process was not available.

Sys_execv

- Replace the currently executing program image with a new process image.
- Process id is unchanged.
- int sys_execv(char *program, char **args)
  - program: path name of the program to run.
  - Args: tf->tf_a0 and tf->tf_a1
- Most of the implementation is already in the runprogram.c
  - Only a few more things.
  - Check the last argument in **args is NULL.
  - Make sure it is less than MAX_ARGS_NUM
  - *copyin* the arguments from user space to kernel space.
  - Create a new address space.
    - *as_create()*
  - Allocate a stack on it.
    - *as_define_stack()*
  - *Copyout* the arguments back onto the new stack

Sys_execv errors

ENODEV	The device prefix of *program* did not exist.
ENOTDIR	A non-final component of *program* was not a directory.
ENOENT	*program* did not exist.
EISDIR	*program* is a directory.
ENOEXEC	*program* is not in a recognizable executable file format, was for the wrong platform, or contained invalid fields.
ENOMEM	Insufficient virtual memory is available.
E2BIG	The total size of the argument strings is too large.
EIO	A hard I/O error occurred.
EFAULT	One of the args is an invalid pointer.

Sys_waitpid

- Wait for the process with pid to exit.

- Return its exit code via the integer pointer *status*.

- pid_t sys_waitpid(pid_t pid, int *status, int options)

- You need a mechanism for processes to show *interest* into each other.

  - You can add restrictions on which processes can show interest.
  - Make sure to prevent deadlocks by either setting restrictions to prevent it or to implement a mechanism to detect it.

- Return the pid with *status* assigned to exit status on success.

- If error, return -1 and set the ret pointer to the error code.

Sys_waitpid errors

EINVAL

The *options* argument requested invalid or unsupported options.

EFAULT

The *status* argument was an invalid pointer.

Sys_exit

- Causes the current process to terminate.

- The process id of the exiting process cannot be reused if there are other processes *interested* in it.

  - Do not put the exited pid back to available pid pool blindly.

- void sys__exit(int code)

  - Code is the exitcode that will be given to other processes who are *interested* in it

Scheduler

- Currently os161 has single queue round-robin scheduler.

- You can modify hardclock.c to have another counter that counts in HZ/2.

- Mostly scheduler.c will be edited.
  - Add a new queue.
  - Add each process a priority and modify make_runnable to match the thread and queue level according to its priority.
  - Modify the scheduler function such that the chances of picking higher level queue will increase.

Testing

- os161/man/testbin has the details about given tests
  - Contains html files
  - Read them carefully and understand what needs to be implemented to pass the tests
  - Be careful: some of them requires VM management to work
- Forktest is very useful
- Also test bin/cp example in the assignment description
- Shell implementation is given but not necessary
  - You can call the tests by p /testbin/forktest
- A basic sys_write is also provided. It will be necessary for printf statements from inside a user-program

Testing

- Build you own tests
- Repeat some of the tests with your new scheduler enabled
  - Report the response times with different quantum sizes
- Make sure to include all the test outputs in your submission

Thank you