Shell

Abhilash Jindal

Agenda

- Understand how shell works (xv6 book Ch0: processes, pipes)
  - Process APIs: fork, wait, exec (OSTEP Ch5)
  - IO redirection, pipes
- Implementations:
  - Fork (xv6 Ch1), copy-on-write (OSTEP Ch23)
  - Exec, loading ELF binaries and running (xv6 Ch2)
  - Pipes, wait, exit (xv6 Ch5)

Shell

- Fork, exec, wait, IO redirection, pipes:
  - ls
  - echo hello > world.txt
  - wc < world.txt
  - echo hello | wc
  - yes OS. In new terminal, ps aux | grep yes; kill <pid>

Fork Semantics

- p1.c: Child gets rc=0 and parents gets child's PID in rc
- fork-cow.c: Address space is copied at the time of fork. Writes of child and parent are not seen to each other
  - Example: each browser tab can be a separate process forked by the main browser
  - Redis forks a backup process to checkpoint in-memory database to disk
- fork-fd.c: OS is maintaining an open file table for each process. Open file table is just an array from file descriptor (integer) to a file struct (containing offsets). Open file table is duplicated at time of fork. When one process writes, the offset is incremented.
  - Example: Browser tabs can write to a common log file without stepping over one another
- dup.c: Duplicate file descriptors can be created within a process. dup copies file descriptors along with offsets. Compare output with nodup.c

Other process APIs and IO redirection

- p2.c: Shell wants to be able to wait for the child process to finish. wait system call does that
- p3.c: Shell does not want to run just copies of itself. It wants to execute other program binaries. exec prepares a new address space where it loads the program binary stored in the ELF format. Exec gives control to the ELF->entry
- p4.c: In the gap between fork and exec, shell can open and close files. By default processes have stdin file (fd=0), stdout file (fd=1), and stderr file (fd=2). Here we close stdout and open “p4.output”. Child will just write to file descriptor 1 which now points to “p4.output”
- pipe.c: pipe returns two file descriptors: a read end and a write end. We close STDOUT and dup the write end. In parent, we close STDIN and dup the read end.

fork in xv6 (proc.c)

- Allocate process: allocate a kernel stack, etc
- Copy memory (copyuvm): copy the page table and all the process’ pages
- Copy trap frame, set return value as 0 in the child process
- Duplicate file descriptors (shared file offset)
- Mark child as runnable

Faster forks

- Addresses may be large => very slow fork
- Use copy-on-write

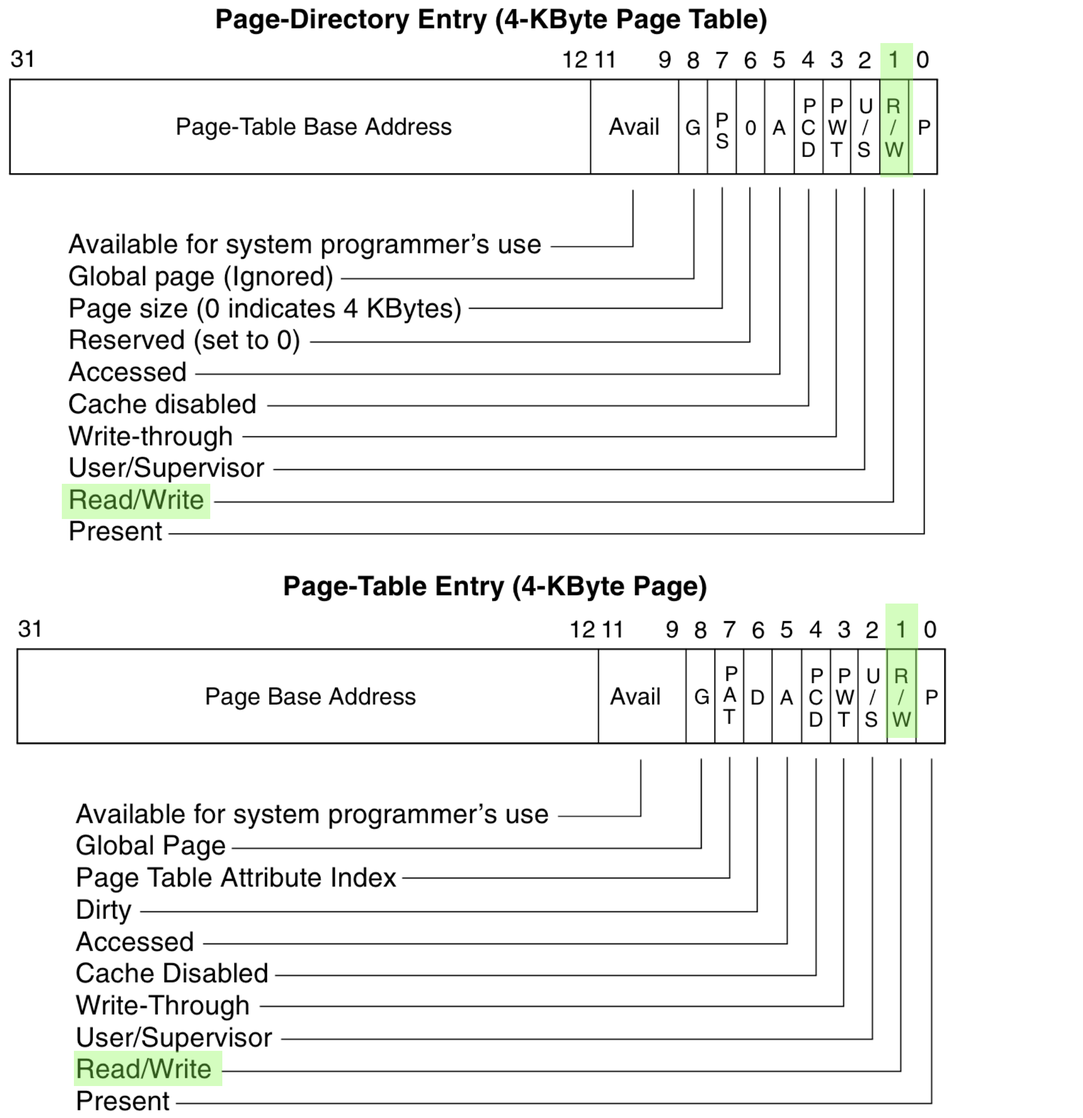
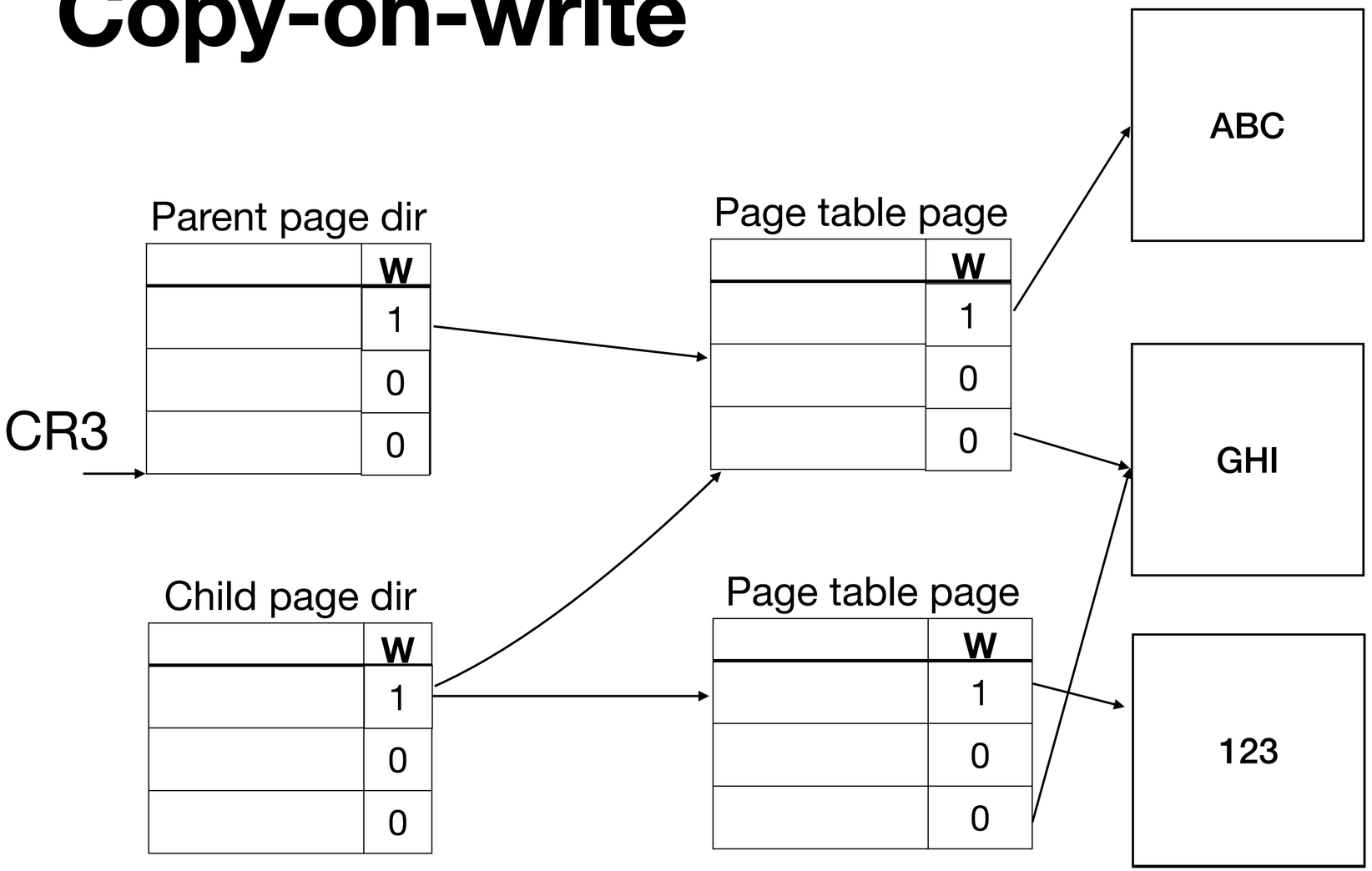


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Copy-on-write



Parent forks

Child process overwrites “ABC” with “123”

exec in xv6 (exec.c)

- Allocate a new page table pgdir=setupkvm()
- Read ELF file to load program from disk to memory
  - Call allocuvm to allocate space and create entries in the page table
  - loaduvm to copy ELF segments
- Add two pages above text and data for stack and guard page
- Remove user flag from the guard page so that if stack is overflowing into data, kernel gets a page fault
- Copy arguments on the stack, mark eip as elf.entry, switch to the new page table, and return (which will return from trap)

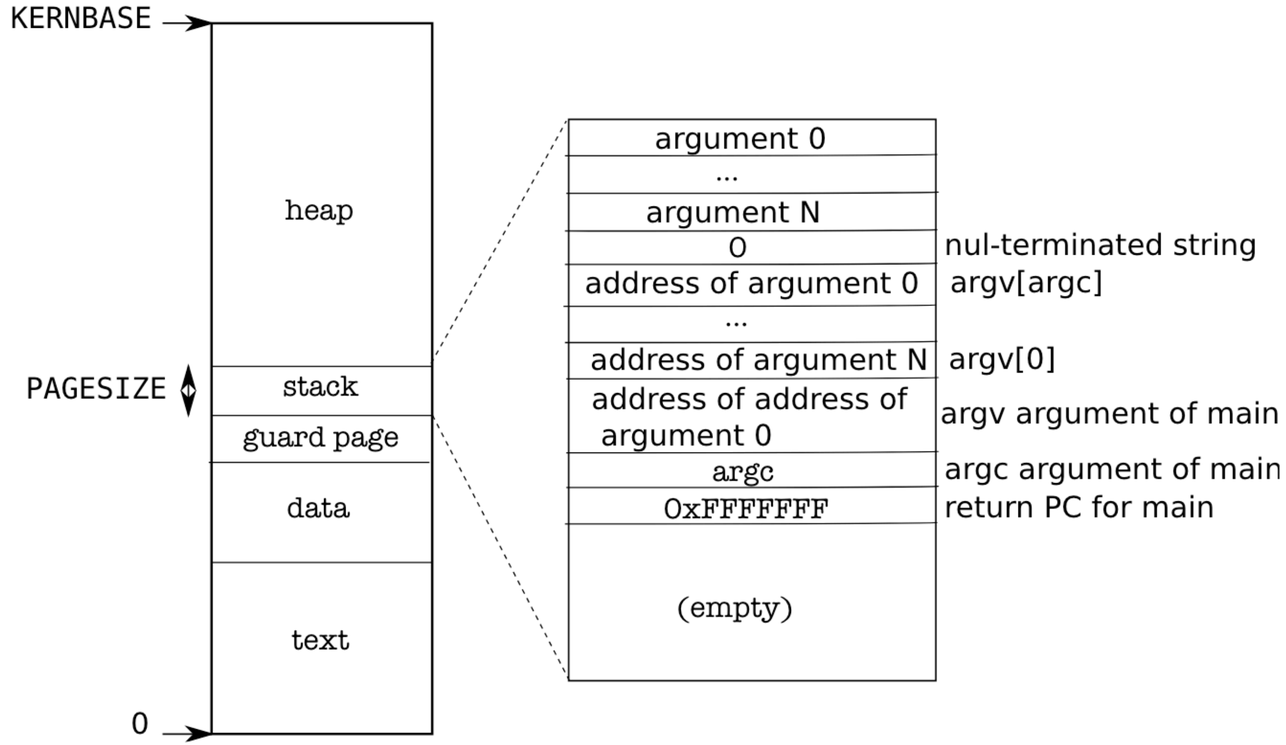


Figure 2-3. Memory layout of a user process with its initial stack.

Executing initial process in xv6 (initcode.S, init.c)

- initcode.S just calls exec with “/init”
- init.c opens the default stdin, stdout, and stderr files. It calls fork and starts shell process. init process itself stays in an infinite loop waiting for child processes to exit
- sh.c is in an infinite loop waiting for user input. It forks+execs process(es) and waits for them to finish before again waiting for user input

exit in xv6 (proc.c)

- When a process ends (see cat.c for example), it calls the exit system call
- exit closes all the open file descriptors, wake up the parent who might be waiting for the child to exit
- If parent exits before the child, it makes initproc as the parent. If child has already exited, it wakes up initproc for cleanup
- Note that wakeup only marks the process and runnable but does not call the scheduler
- Finally, the process marks itself as zombie and calls the scheduler

wait in xv6 (proc.c)

- wait cleans up the process states. It returns after cleaning up a child process
- It loops over process table to find out a zombie child process and cleans up their kernel stack, page table, and finally mark the process slot as UNUSED
- If it has running children (not ZOMBIE), it goes into sleep to be woken up whenever a child process exits

kill in xv6 (kill.c, proc.c)

- kill(pid): we do not want to immediately kill a process. It may be in middle of something (like writing disk blocks)
- kill(pid) only marks p->killed=1 and marks it as runnable if it was sleeping
- After waking up, we usually check if the process is already killed and gracefully clean up (see console.c, pipe.c, sysproc.c)
- When entering the OS or when returning to process, if process is killed, we call exit (see trap.c). These are good places to exit since kernel was not in the middle of servicing requests from the process.

pipe in xv6 (pipe.c)

- See “case PIPE” in sh.c. It calls pipe sys call and passes it an integer pointer pointing to two integers
- sys_pipe (sysfile.c) calls pipealloc which returns two file descriptors: a read end of the pipe and a write end of the pipe and adds the descriptors to open file table of the current process (shell) and writes the two file descriptors in the integers pointed to by the argument
- Pipe behaves just like files: fileread/filewrite/fileclose is forwarded to piperead/pipewrite/pipeclose (see file.c)
- sh.c forks the reader (writer) process, closes its stdin (stdout), and duplicates read side (write side) of the pipe and closes p[0] and p[1] descriptors in all processes: reader/writer/shell.
- pipealloc (in pipe.c) allocates one page to hold the pipe struct which has 512 bytes of “data”. Data will be transferred between processes via this “data” (i.e, without going through disk)
- Try to understand rest of pipe.c. We will look at it when we talk about locks!