Administrivia

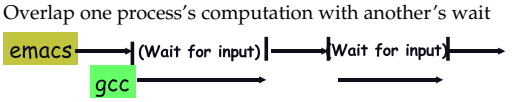
- **Lab 1 is up, Due Thursday, Jan. 22**
  - Due at start of lecture – 4:15pm
  - Free extension to midnight if you come to lecture
  - Or for SCPD students if you watch lecture that day
- **No credit for late assignments w/o extension**
- **Ask cs140-staff for extension if you can't finish**
  - Tell us where you are with the project
  - And how much longer you need to finish
- **We are nice people**
  - ...but can't help you if you miss deadlines and don't talk to us
- **Section tomorrow, 4:15pm, Gates B01**

Processes

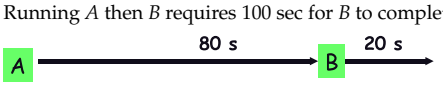
- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Examples (can all run simultaneously):**
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `emacs` – text editor
  - `firefox` – web browser
- **Non-examples (implemented as one process):**
  - Multiple firefox windows or emacs frames (still one process)
- **Why processes?**
  - Simplicity of programming
  - Higher throughput (better CPU utilization), lower latency

Speed

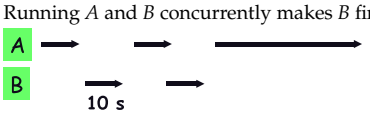
- **Multiple processes can increase CPU utilization**
  - Overlap one process's computation with another's wait



- **Multiple processes can reduce latency**
  - Running *A* then *B* requires 100 sec for *B* to complete



  - Running *A* and *B* concurrently makes *B* finish faster



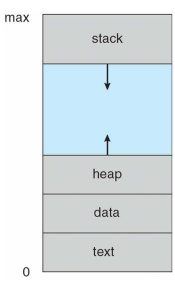
  - *A* slightly slower, but less than 100 sec unless *A* and *B* both completely CPU-bound

Processes in the real world

- **Processes, parallelism fact of life much longer than OSes have been around**
  - E.g., say takes 1 worker 10 months to make 1 widget
  - Company may hire 100 workers to make 10,000 widgets
  - Latency for first widget $\gg$ 1/10 month
  - Throughput may be $<$ 10 widgets per month (if can't perfectly parallelize task)
  - Or $>$ 10 widgets per month if better utilization (e.g., 100 workers on 10,000 widgets never idly waiting for paint to dry)
- **You will see this with Pintos**
  - Don't expect labs to take 1/3 time with three people

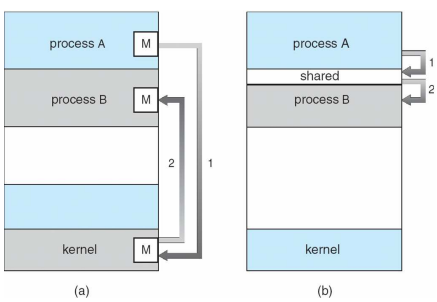
A process's view of the world

- **Each proc. P_i has own view of machine**
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)
- `*(char *)0xc000` **different in P_1 & P_2**
- **Greatly simplifies programming model**
  - `gcc` does not care that `firefox` is running



- **Sometimes want interaction between processes**
  - Simplest is through files: `emacs` edits file, `gcc` compiles it
  - More complicated: Shell/command, Window manager/app.

Inter-Process Communication



- **How can processes interact in real time?**
  (a) By passing messages through the kernel
  (b) By sharing a region of physical memory
  (c) Through asynchronous signals or alerts

Rest of lecture

- **User view of processes**
  - Crash course in basic Unix/Linux system call interface
  - How to create, kill, and communicate between processes
- **Kernel view of processes**
  - Implementing processes in the kernel
- **Threads**
- **How to implement threads**

UNIX files I/O

- **Applications "open" files (or devices) by name**
  - I/O happens through open files
- `int open(char *path, int flags, /*mode*/...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout, printf` in ANSI C)
  - 2 – "standard error" (`stderr, perror` in ANSI C)
  - Normally all three attached to terminal
- **Example:** `type.c`
  - Prints the contents of a file to `stdout`

`type.c`

```
void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}
```

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new proc. in "parent"
  - Returns 0 in "child"
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for "cleanup")
  - `SIGKILL` stronger, kills process always

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
  - `int execvp (char *prog, char **argv);`
    Search `PATH` for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    List arguments one at a time, finish with `NULL`
- **Example:** `minish.c`
  - Loop that reads a command, then executes it
- **Warning: Pintos** `exec` **more like combined fork/exec**

`minish.c` (simplified)

```
pid_t pid; char **av;

void doexec () {
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
  /* ... main loop: */
  for (;;) {
    parse_next_line_of_input (&av, stdin);

    switch (pid = fork ()) {
    case -1:
      perror ("fork"); break;
    case 0:
      doexec ();
    default:
      waitpid (pid, NULL, 0); break;
    }
  }
```


Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
  - Sets *close on exec* flag if `val` = 1, clears if `val` = 0
  - Makes file descriptor non-inheritable by spawned programs
- **Example:** `redirsh.c`
  - Loop that reads a command and executes it
  - Recognizes `command < input > output 2> errlog`

skip 3 slides→

`redirsh.c`

```
void doexec (void) {
  int fd;

  /* infile non-NULL if user typed "command < infile" */
  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  /* ... Do same for outfile -> fd 1, errfile -> fd 2 ... */

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Writes to `fds[1]` will be read on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read/write/close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`, or if blocked
    - Fails with EPIPE
- **Example:** `pipesh.c`
  - Sets up pipeline `command1 | command2 | command3 ...`

`pipesh.c` (simplified)

```
void doexec (void) {
  int pipefds[2];
  while (outcmd) {
    pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break;
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&av, &outcmd, outcmd);
      break;
    }
  }
  /* ... */
}
```


Why fork?

- **Most calls to `fork` followed by `execve`**
- **Could also combine into one *spawn* system call**
  - This is what Pintos `exec` does
- **Occasionally useful to fork one process**
  - Unix *dump* utility backs up file system to tape
  - If tape fills up, must restart at some logical point
  - Implemented by forking to revert to old state if tape ends
- **Real win is simplicity of interface**
  - Tons of things you might want to do to child:
    Manipulate file descriptors, environment, resource limits, etc.
  - Yet `fork` requires *no* arguments at all

Spawning process w/o fork

- **Without fork, require tons of different options**
- **Example: Windows `CreateProcess` system call**

BOOL CreateProcess(
LPCTSTR *lpApplicationName*, // pointer to name of executable module
LPTSTR *lpCommandLine*, // pointer to command line string
LPSECURITY_ATTRIBUTES *lpProcessAttributes*, // process security attr.
LPSECURITY_ATTRIBUTES *lpThreadAttributes*, // thread security attr.
BOOL *bInheritHandles*, // handle inheritance flag
DWORD *dwCreationFlags*, // creation flags
LPVOID *lpEnvironment*, // pointer to new environment block
LPCTSTR *lpCurrentDirectory*, // pointer to current directory name
LPSTARTUPINFO *lpStartupInfo*, // pointer to STARTUPINFO
LPPROCESS_INFORMATION *lpProcessInformation* // pointer to PROCESS_INFORMATION **);**

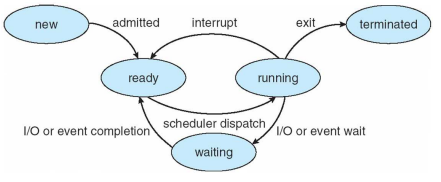
Implementing processes

- **OS keeps data structure for each proc**
  - Process Control Block (PCB)
- **Tracks *state* of the process**
  - Running, runnable, blocked, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether proc. being debugged, system call binary emulation in use, ...

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

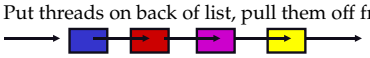
Process states



- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different proc. to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop, if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

Scheduling

- **How to pick which process to run**
- **Scan process table for first runnable?**
  - Expensive. Weird priorities (small pid's better)
  - Divide into runnable and blocked processes
- **FIFO?**
  - Put threads on back of list, pull them off from front



(pintos does this: thread.c)
- **Priority?**
  - Give some threads a better shot at the CPU

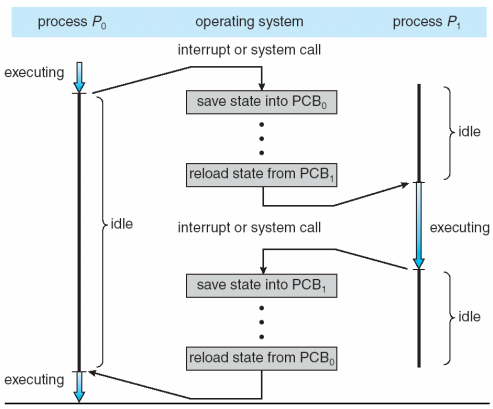
Scheduling policy

- **Want to balance multiple goals**
  - *Fairness* – don't starve processes
  - *Priority* – reflect relative importance of procs
  - *Deadlines* – must do *x* (play audio) by certain time
  - *Throughput* – want good overall performance
  - *Efficiency* – minimize overhead of scheduler itself
- **No universal policy**
  - Many variables, can't optimize for all
  - Conflicting goals (e.g., throughput or priority vs. fairness)
- **We will spend a whole week on this topic**

Preemption

- **Can preempt running proc. when kernel gets control**
- **Running proc. can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current proc. to sleep—e.g., read from disk
  - May make other proc. runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running proc. used up quantum, schedule another
- **Device interrupt**
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- **Changing running proc. is called a *context switch***

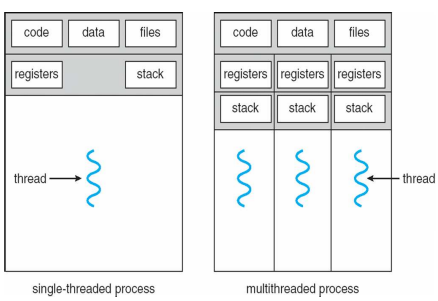
Context switch



Context switch details

- **Very machine dependent. Typical things include:**
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations
- **Non-negligible cost**
  - Save/restore floating point registers expensive
    optimization: only save if proc. used floating point
  - May require flushing TLB (memory translation hardware)
    optimization: don't flush kernel's own data from TLB
  - Usually causes more cache misses (switch working sets)

Threads



- **A thread is a schedulable execution context**
  - Program counter, stack, registers, . . .
- **Simple programs use one thread per process**
- **But can also have multi-threaded programs**
  - Multiple threads running in same process's address space

Why threads?

- **Most popular abstraction for concurrency**
  - Lighter-weight abstraction than processes
  - All threads in one process share memory, file descriptors, etc.
- **Allows one process to use multiple CPUs or cores**
- **Allows program to overlap I/O and computation**
  - Same benefit as OS running `emacs` & `gcc` simultaneously
  - E.g., threaded web server services clients simultaneously:

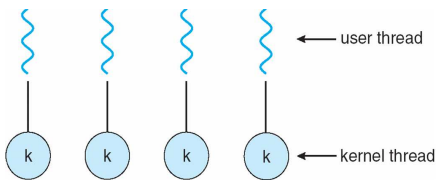
```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

- **Most kernels have threads, too**
  - Typically at least one kernel thread for every process

Thread package API

- `tid create (void (*fn) (void *), void *arg);`
  - Create a new thread, run `fn` with `arg`
- `void exit ();`
  - Destroy current thread
- `void join (tid thread);`
  - Wait for thread `thread` to exit
- **Plus lots of support for synchronization [next week]**
- **Can have preemptive or non-preemptive threads**
  - Preemptive causes more race conditions
  - Non-preemptive can't take advantage of multiple CPUs
  - Before prevalent SMPs, most kernels non-preemptive

Kernel threads

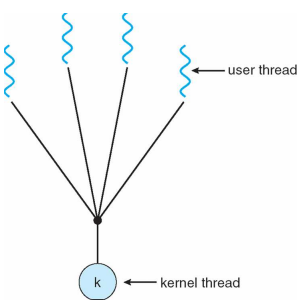


- **Can implement thread `create` as system call**
- **Start with process abstraction in kernel**
- **Strip out unnecessary features**
  - Same address space, file table, etc.
  - `rfork/clone` actually allow individual control
- **Faster than a process, but still very heavy weight**

Why kernel threads suck

- **Every thread operation must go through kernel**
  - create, exit, join, synchronize, or switch for any reason
  - On Athlon 3400+: syscall takes 359 cycles, fn call 6 cycles
  - Result: threads 10x-30x slower when implemented in kernel
- **One-size fits all thread implementation**
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need
- **General heavy-weight memory requirements**
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

User threads



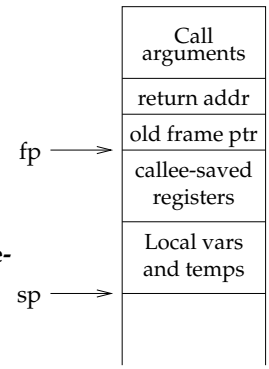
- **An alternative: implement in user-level library**
  - One kernel thread per process
  - `create`, `exit`, etc., just library functions

Implementing user-level threads

- **Allocate a new stack for reach thread `create`**
- **Keep a queue of runnable threads**
- **Replace networking system calls (read/write/etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (`setitimer`)**
  - Switch to another thread on timer signals (preemption)
- **Multi-threaded web server example**
  - `read` data from connecting web browser
  - No data? replaced `read` schedules another thread
  - On timer or idle check which connections have new data
- **How to switch threads?**

Background: calling conventions

- ***sp* register always base of stack**
  - frame pointer (*fp*) is old *sp*
- **Local vars in stack & registers**
  - By convention, registers divided into caller- and callee-saved
- **Function arguments go in callee-saved regs and on stack**



Background: procedure calls

save active caller registers
call foo → saves used callee registers
...do stuff...
restores callee registers
jumps back to pc
restore caller regs

- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state saved not saved**
  - Callee-saved regs, global variables, stack pointer

Threads vs. procedures

- **Threads may resume out of order:**
  - Cannot use LIFO stack to save state
  - General solution: one stack per thread
- **Threads switch less often:**
  - Don't partition registers (why?)
- **Threads involuntarily interrupted:**
  - Synchronous: procedure call can use compiler to save state
  - Asynchronous: thread switch code saves all registers
- **More than one than one thread can run**
  - Scheduling: what to run next and on which CPU?
  - Procedure call scheduling obvious: run called procedure

Example user threads implementation

- **Per-thread state in thread control block structure**

```
typedef struct tcb {
  unsigned long md_esp;         /* Stack pointer of thread */
  char *t_stack;                /* Bottom of thread's stack */
  /* ... */
};
```

- **Machine-dependent thread-switch function:**
  - `void thread_md_switch (tcb *current, tcb *next);`
- **Machine-dependent thread initialization function:**
  - `void thread_md_init (tcb *t,`
    `void (*fn) (void *), void *arg);`

i386 thread_md_switch

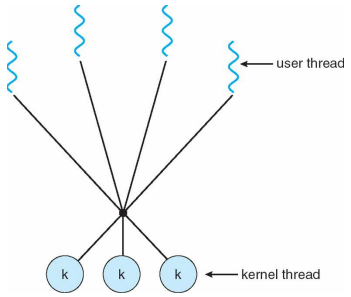
```
pushl %ebp; movl %esp,%ebp          # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi  # Save callee-saved regs

movl 8(%ebp),%edx       # %edx = thread_current
movl 12(%ebp),%eax      # %eax = thread_next
movl %esp,(%edx)        # %edx->md_esp = %esp
movl (%eax),%esp        # %esp = %eax->md_esp

popl %edi; popl %esi; popl %ebx     # Restore callee saved regs
popl %ebp                           # Restore frame pointer
ret                                 # Resume execution
```

- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
- **You will see very similar code in Pintos switch.S**

Why user threads suck

- **Can't take advantage of multiprocessors**
- **A blocking system call blocks all threads**
  - Can replace read to handle network connections
  - But usually OSes don't let you do this for disk
  - So uncached disk read blocks all processes
- **A page fault blocks all threads**
- **Hard to run as many threads as CPUs**
  - Don't know how many CPUs available
  - Don't know which threads are blocked
- **Possible deadlock if one thread blocks on another**
  - May block entire process and make no progress
  - [More on deadlock next week.]

Uthreads on kthreads



- **User threads implemented on kernel threads**
  - Multiple kernel threads per process
  - `create`, `exit`, etc., still library functions as before

Problems

- **Still many of the same problems as before**
- **Hard to keep same # ktrheads as available CPUs**
  - Kernel knows how many CPUs available
  - But tries to hide this from applications with preemption
- **Kernel doesn't know about relative importance of threads**
  - Might preempt kthread in which library holds important lock

Lessons

- **Threads best implemented as a library**
  - But kernel threads not the best interface on which to build this
- **Better kernel interfaces have been suggested**
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features)
- **Today shouldn't dissuade you from using threads**
  - Standard user or kernel threads are fine for most purposes
  - Use kernel threads if I/O concurrency main goal
  - Use user on kernel threads for highly concurrent (e.g,. scientific applications) with many thread switches
- **...though the next two lectures may**
  - Concurrency greatly increases the complexity of a program!
  - Leads to all kinds of nasty race conditions

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}

int
main (int argc, char **argv)
{
  int argno;
  for (argno = 1; argno < argc; argno++)
    typefile (argv[argno]);
  exit (0);
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    avreserve (n);
    av[n] = a;
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
```

```
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
char *infile;
char *outfile;
char *errfile;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  infile = outfile = errfile = NULL;
  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    if (a[0] == '<')
      infile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '>')
      outfile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '2' && a[1] == '>')
      errfile = a[2] ? a + 2 : strtok (NULL, " \t\r\n");
    else {
      avreserve (n);
      av[n] = a;
    }
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  int fd;

  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
```

```
      dup2 (fd, 0);
      close (fd);
    }
  }

  if (outfile) {
    if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 1) {
      dup2 (fd, 1);
      close (fd);
    }
  }

  if (errfile) {
    if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 2) {
      dup2 (fd, 2);
      close (fd);
    }
  }

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
char *infile;
char *outfile;
char *errfile;
char *outcmd;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  outcmd = infile = outfile = errfile = NULL;
  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    if (a[0] == '<')
      infile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '>')
      outfile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '|') {
      if (!a[1])
        outcmd = strtok (NULL, "");
      else {
        outcmd = a + 1;
        a = strtok (NULL, "");
        while (a > outcmd && !a[-1])
          *--a = ' ';
      }
    }
    else if (a[0] == '2' && a[1] == '>')
      errfile = a[2] ? a + 2 : strtok (NULL, " \t\r\n");
    else {
      avreserve (n);
      av[n] = a;
    }
    a = strtok (NULL, " \t\r\n");
  }
}
```

```
void
doexec (void)
{
  int fd;
  int pipefds[2];

  while (outcmd) {
    if (outfile) {
      fprintf (stderr, "syntax error: > in pipe writer\n");
      exit (1);
    }

    if (pipe (pipefds) < 0) {
      perror ("pipe");
      exit (0);
    }

    switch (fork ()) {
    case -1:
      perror ("fork");
      exit (1);
    case 0:
      if (pipefds[1] != 1) {
        dup2 (pipefds[1], 1);
        close (pipefds[1]);
      }
      close (pipefds[0]);
      outcmd = NULL;
      break;
    default:
      if (pipefds[0] != 0) {
        dup2 (pipefds[0], 0);
        close (pipefds[0]);
      }
      close (pipefds[1]);
      parseline (outcmd);
      if (infile) {
        fprintf (stderr, "syntax error: < in pipe reader\n");
        exit (1);
      }
      break;
    }
  }

  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  if (outfile) {
    if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 1) {
      dup2 (fd, 1);
      close (fd);
    }
  }
```

```
  if (errfile) {
    if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (errfile);
      exit (1);
    }
    if (fd != 2) {
      dup2 (fd, 2);
      close (fd);
    }
  }

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```