Administrivia

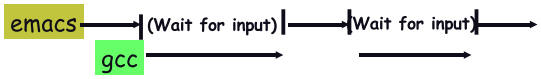
- **Section scheduled at 2:15pm in Gates B03 (next door)**
  - Please attend this Friday to learn about lab 1
- **Lab 1 will be distributed Friday**
  - Already on-line, but we are beta testing it
  - Due Friday Jan. 23 at noon
- **Ask `cs140-staff` for extension if you can't finish**
  - Tell us where you are with the project,
  - How much more you need to do, and
  - How much longer you need to finish
- **No credit for late assignments w/o extension**
- **If you are enrolled pass/fail, please disclose to your partners**
  - We don't recommend pass/fail, but it is allowed

Processes

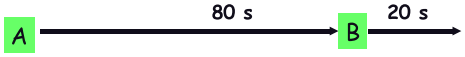
- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Examples (can all run simultaneously):**
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `emacs` – text editor
  - `firefox` – web browser
- **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

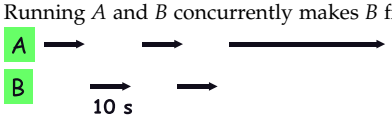
- **Multiple processes can increase CPU utilization**
  - Overlap one process's computation with another's wait



- **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



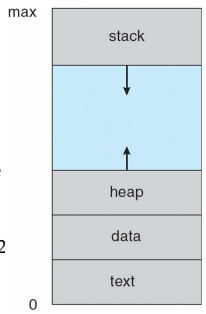
  - A is slower than if it had whole machine to itself, but still < 100 sec unless both A and B completely CPU-bound

Processes in the real world

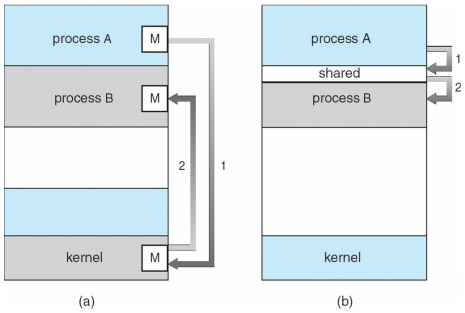
- **Processes, parallelism fact of life much longer than OSes have been around**
  - E.g., say takes 1 worker 10 months to make 1 widget
  - Company may hire 100 workers to make 100 widgets
  - Latency for first widget $\gg 1/10$ month
  - Throughput may be < 10 widgets per month (if can't perfectly parallelize task)
  - And 100 workers making 10,000 widgets may achieve > 10 widgets/month (e.g., if workers never idly wait for paint to dry)
- **You will see these effects in you Pintos project group**
  - May block waiting for partner or need time to coordinate
  - Labs won't take 1/3 time with three people

A process's view of the world

- **Each process 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**
- **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
  - Running example: how to implement a shell
- **Kernel view of processes**
  - Implementing processes in the kernel
- **Threads**
- **How to implement threads**

Outline

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process 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`

redirsh.c

```
void doexec (void) {
  int fd;
  if (infile) {     /* non-NULL for "command < 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 signal ignored, fails with EPIPE
- **Example:** `pipesh.c`
  - Sets up pipeline `command1 | command2 | command3 ...`

pipesh.c (simplified)

```
void doexec (void) {
  while (outcmd) {
    int pipefds[2]; 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**
  - Also `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`, ...

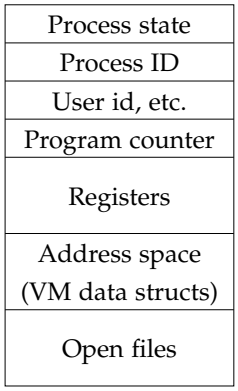
```
BOOL WINAPI CreateProcess(
  _In_opt_      LPCTSTR lpApplicationName,
  _Inout_opt_   LPTSTR lpCommandLine,
  _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_          BOOL bInheritHandles,
  _In_          DWORD dwCreationFlags,
  _In_opt_      LPVOID lpEnvironment,
  _In_opt_      LPCTSTR lpCurrentDirectory,
  _In_          LPSTARTUPINFO lpStartupInfo,
  _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```

Outline

1. User view of processes
2. Kernel view of processes
3. Threads
4. How to implement threads

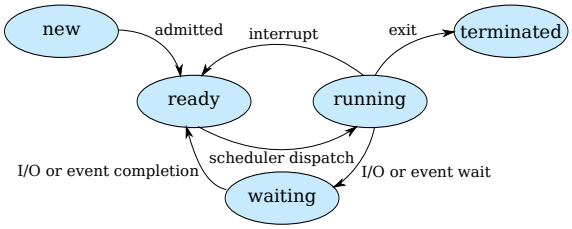
Implementing processes

- **OS keeps data structure for each proc**
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux, and just `struct thread` in Pintos
- **Tracks *state* of the process**
  - Running, ready (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 being debugged, which system call binary emulation in use, ...



PCB

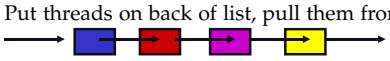
Process states



- **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 process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop (or halt CPU), 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 pids do better)
  - Divide into runnable and blocked processes
- **FIFO?**
  - Put threads on back of list, pull them from front



(pintos does this: `thread.c`)

- **Priority?**
  - Give some threads a better shot at the CPU

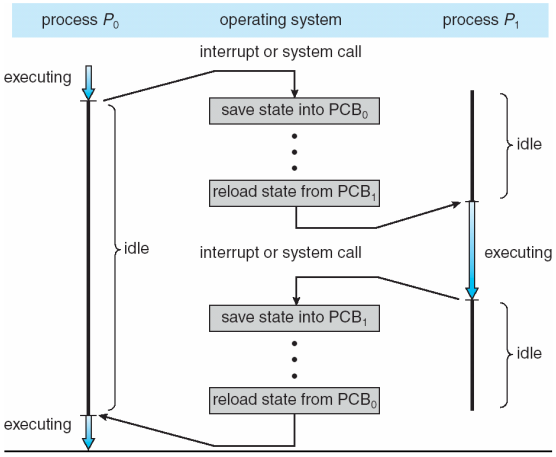
Scheduling policy

- **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 lecture on this topic**

Preemption

- **Can preempt a process when kernel gets control**
- **Running process can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running process 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 process is called a *context switch***

Context switch

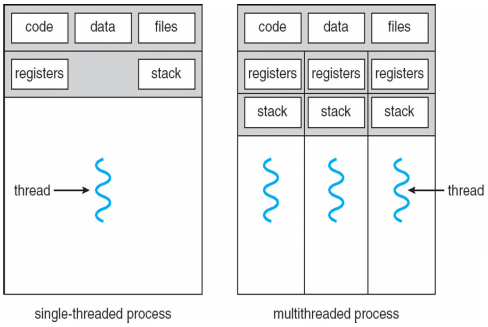


Context switch details

- **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 process used floating point
  - May require flushing TLB (memory translation hardware)
    - HW Optimization 1: don't flush kernel's own data from TLB
    - HW Optimization 2: use tag to avoid flushing any data
  - Usually causes more cache misses (switch working sets)

Outline

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:

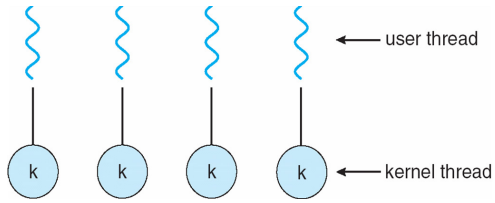
```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

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

Thread package API

- `tid thread_create (void (*fn) (void *), void *);`
  - Create a new thread, run `fn` with `arg`
- `void thread_exit ();`
  - Destroy current thread
- `void thread_join (tid thread);`
  - Wait for thread `thread` to exit
- **Plus lots of support for synchronization [in 3 weeks]**
- **See [Birell] for good introduction**
- **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

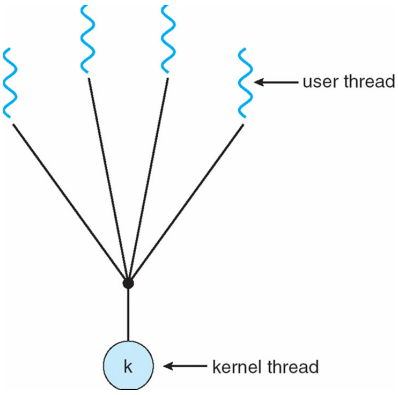


- **Can implement `thread_create` as a system call**
- **To add `thread_create` to an OS that doesn't have it:**
  - Start with process abstraction in kernel
  - `thread_create` like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - `rfork`/`clone` syscalls actually allow individual control
- **Faster than a process, but still very heavy weight**

Limitations of kernel-level threads

- **Every thread operation must go through kernel**
  - create, exit, join, synchronize, or switch for any reason
  - On my laptop: syscall takes 100 cycles, fn call 5 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
  - `thread_create`, `thread_exit`, etc., just library functions

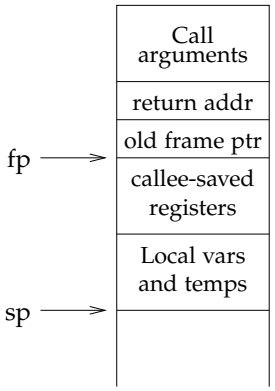
Implementing user-level threads

- **Allocate a new stack for each `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**
  - Thread calls `read` to get data from remote web browser
  - "Fake" `read` *function* makes `read` *syscall* in non-blocking mode
  - No data? schedule another thread
  - On timer or when idle check which connections have new data

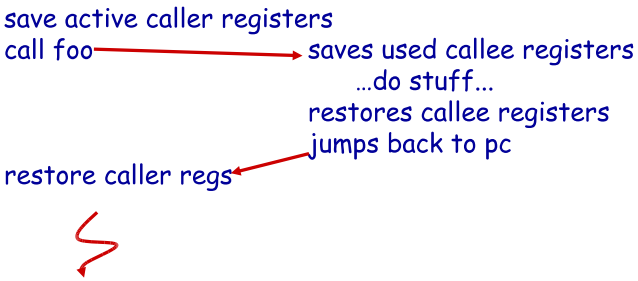
Outline

Background: calling conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs (`%eax` [return val], `%edx`, & `%ecx` on x86)
  - But must restore *callee-saved* ones to original value upon return (on x86, `%ebx`, `%esi`, `%edi`, plus `%ebp` and `%esp`)
- ***sp* register always base of stack**
  - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in caller-saved regs and on stack**
  - With x86, all arguments on stack



Background: procedure calls



- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state 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 can be 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 at a time:**
  - Procedure call scheduling obvious: Run called procedure
  - Thread scheduling: What to run next and on which CPU?

Example user threads implementation

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

```
typedef struct tcb {
  uintptr_t 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

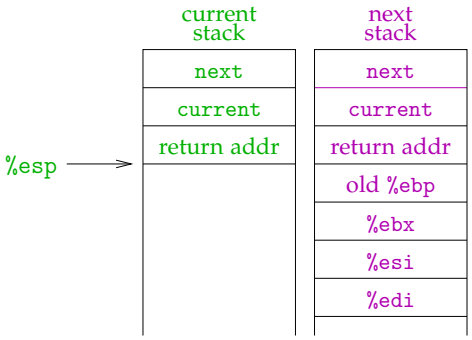
```
pushl %ebp; movl %esp,%ebp          # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi  # Save callee-saved regs

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
```

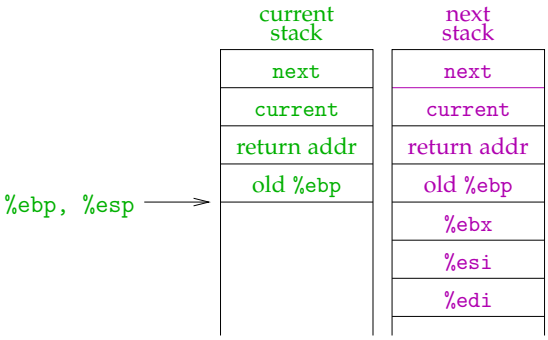
- **This is literally switch code from simple thread library**
  - Nothing magic happens here
  - You will see very similar code in Pintos `switch.S`

i386 thread_md_switch



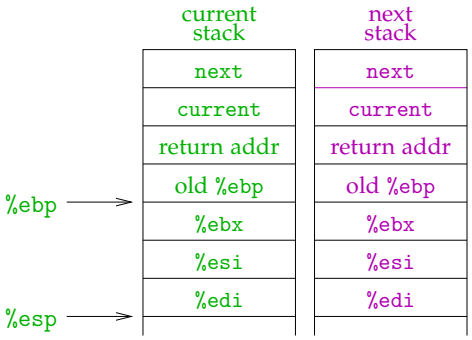
- **This is literally switch code from simple thread library**
  - Nothing magic happens here
  - You will see very similar code in Pintos `switch.S`

i386 thread_md_switch



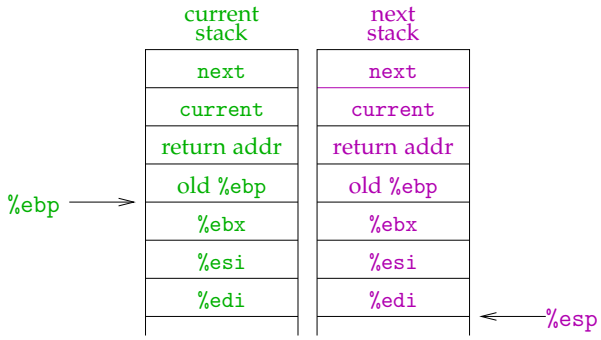
- **This is literally switch code from simple thread library**
  - Nothing magic happens here
  - You will see very similar code in Pintos `switch.S`

i386 thread_md_switch



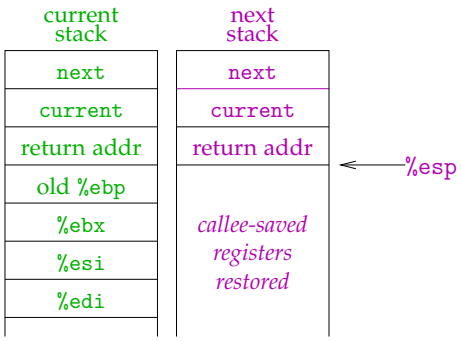
- **This is literally switch code from simple thread library**
  - Nothing magic happens here
  - You will see very similar code in Pintos `switch.S`

i386 thread_md_switch



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

i386 thread_md_switch

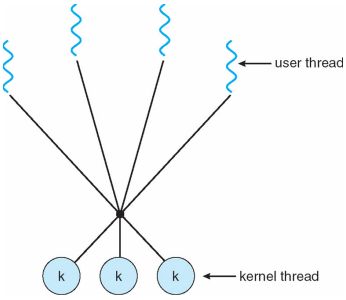


- **This is literally switch code from simple thread library**
  - Nothing magic happens here
  - You will see very similar code in Pintos `switch.S`

Limitations of user-level threads

- **Can't take advantage of multiple CPUs or cores**
- **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 one uncached disk read blocks all threads
- **A page fault blocks all threads**
- **Possible deadlock if one thread blocks on another**
  - May block entire process and make no progress
  - [More on deadlock in future lectures.]

User threads on kernel threads



- **User threads implemented on kernel threads**
  - Multiple kernel-level threads per process
  - `thread_create`, `thread_exit` still library functions as before
- **Sometimes called $n : m$ threading**
  - Have n user threads per m kernel threads
    (Simple user-level threads are $n : 1$, kernel threads $1 : 1$)

Limitations of $n : m$ threading

- **Many of same problems as $n : 1$ threads**
  - Blocked threads, deadlock, ...
- **Hard to keep same # ktrheads as available CPUs**
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - But tries to hide these things from applications for transparency
  - So user-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- **Kernel doesn't know relative importance of threads**
  - Might preempt kthread in which library holds important lock

Lessons

- **Threads best implemented as a library**
  - But kernel threads not best interface on which to do 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, now Windows is trying it)
- **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 $n : m$ threads for highly concurrent (e.g,. scientific applications) with many thread switches
- **. . . though concurrency/synchronization 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 <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);
  if (avsize <= n)
    avsize = n + 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);
  if (avsize <= n)
    avsize = n + 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);
  if (avsize <= n)
    avsize = n + 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;

  while (outcmd) {
    int pipefds[2];

    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;
    }
  }
}
```