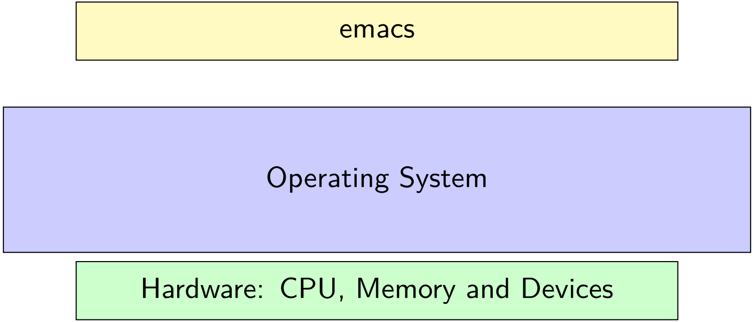
CS350: Operating Systems

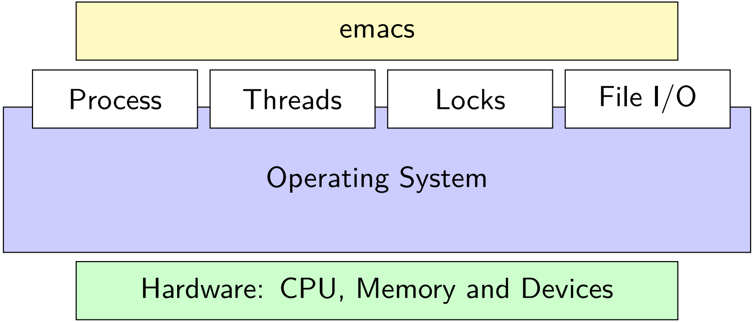
Ali Mashtizadeh

University of Waterloo

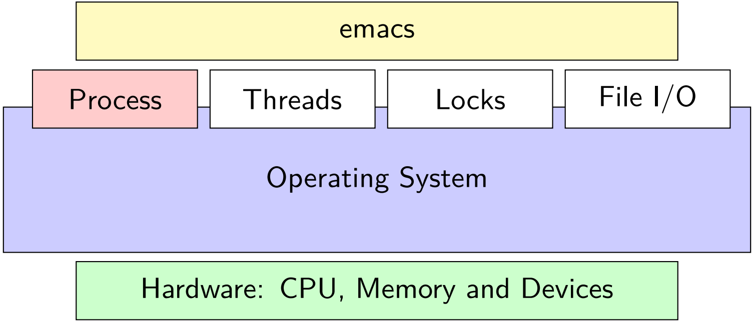
Operating System



Operating System: Basic Abstractions and APIs



Today: Introduce the Process Abstraction

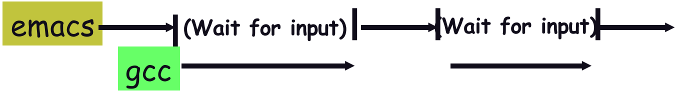


Processes

- A *process* is an instance of a program running
- 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)
- Modern OSes run multiple processes simultaneously
- Why processes?
  - Simplicity of programming
  - Higher throughput (better CPU utilization), lower latency

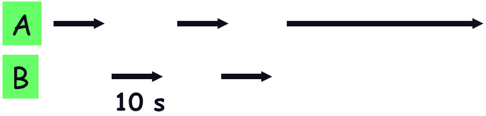
Speed

- 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



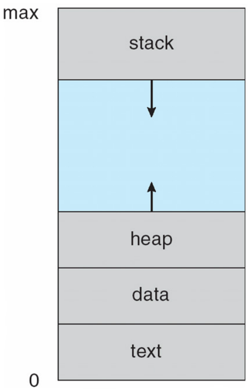
  - A is slower than if it had whole machine to itself, but still < 100 sec unless both A and B completely CPU-bound

Concurrency and parallelism

- 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
- Most computers, laptops, and phones are multi-core!
- Computer with 4 cores can run 4 processes in parallel
- Result: $4\times$ throughput

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.



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

Outline

1. User view of processes
2. Kernel view of processes

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);`
  - Wait for a child process to terminate
  - `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);`
  - Execute a new program
  - `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

minish.c (simplified)

Parent Process (PID 5)

```c
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```

minish.c (simplified)

Parent Process (PID 5)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```

minish.c (simplified)

Parent Process (PID 5)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: // ← After Fork (pid = 6)
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0: // ← After Fork
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```

minish.c (simplified)

Parent Process (PID 5)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: // ← After Fork (pid = 6)
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av); // ← After For
  perror(av[0]); // Never executes!
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```

minish.c (simplified)

Parent Process (PID 5)

```c
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: // ← After Fork (pid = 6)
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

- Replaced by the new program

```c
int
main(int argc, const char *argv[])
{
  // ← Starts here!
  ...
  exit(0);
}
```

minish.c (simplified)

Parent Process (PID 5)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
        // ← waitpid returns
      }
    }
```

Child Process (PID 6)

- Replaced by the new program

```
int
main(int argc, const char *argv[])
{

  ...
  exit(0); // ← Wake up waitpid
}
```

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)
- Example: `redirsh.c`
  - Loop that reads a command and executes it
  - Recognizes `command < input > output 2> errlog`

redirsh.c

```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_input(&av, &outcmd, outcmd);
      break;
    }
  }
```

Why fork?

- Most calls to `fork` followed by `execve`
- Could also combine into one *spawn* system call
- Occasionally useful to fork one process
  - Pre-forked Webservers for parallelism
  - Creates one process per core to serve clients
  - Lots of uses: Nginx, PostgreSQL, etc.
- 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

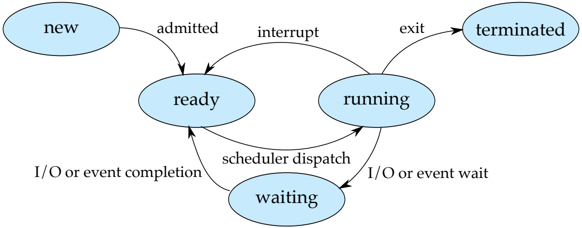
Implementing processes

- OS keeps data structure for each proc
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux, and `struct Process` in COS
- 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, . . .

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 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/Round-Robin?
  - Put threads on back of list, pull them from front

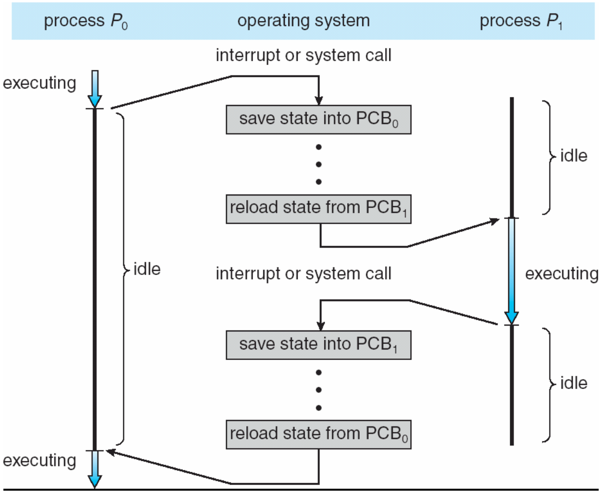
    

    (COS `sys/kern/sched.c`)
- Priority?
  - Give some threads a better shot at the CPU

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)