CS 318 Principles of Operating Systems

Fall 2022

Lecture 3: Processes

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

Lab 0

- Due this Thursday
- Done individually (***cannot*** share with or copy from your to-be-teammates)

Find your project group member soon

- So you can get started with Lab 1 without delay
- Fill out Google form of group info

Recap: Architecture Support for OS

Manipulating privileged machine state

- CPU protection: dual-mode operation, protected instructions
- Memory protection: MMU, virtual address

Generating and handling "events"

- Interrupt, syscall, trap
- Interrupt controller, IVT
- Fix fault vs. notify proceed

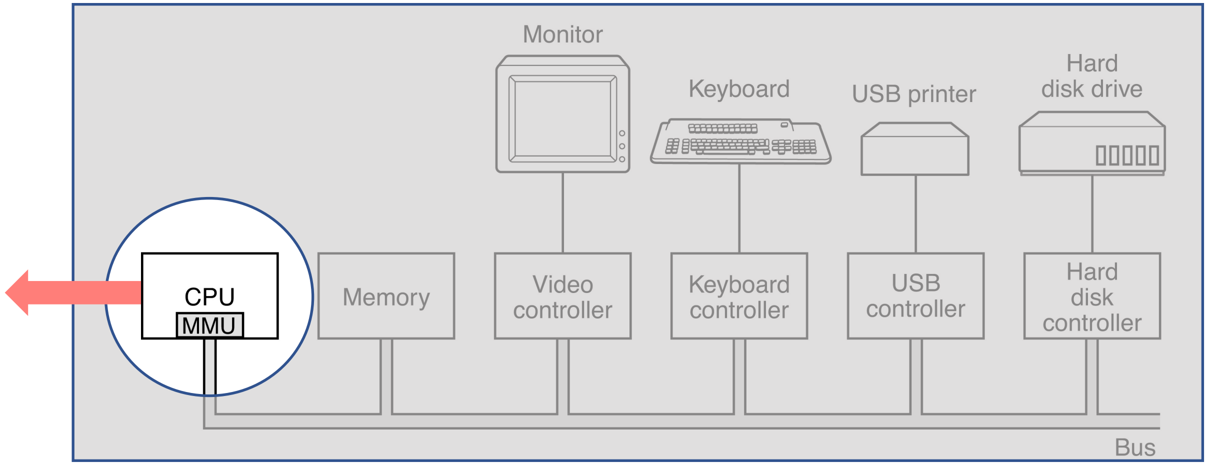
	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

Mechanisms to handle concurrency

- Interrupts, atomic instructions

Overview

OS abstraction?



Today's topics are processes and process management

- What are the units of execution?
- How are those units of execution represented in the OS?
- How is work scheduled in the CPU?
- What are the possible execution states of a process?
- How does a process move from one state to another?

Process Abstraction

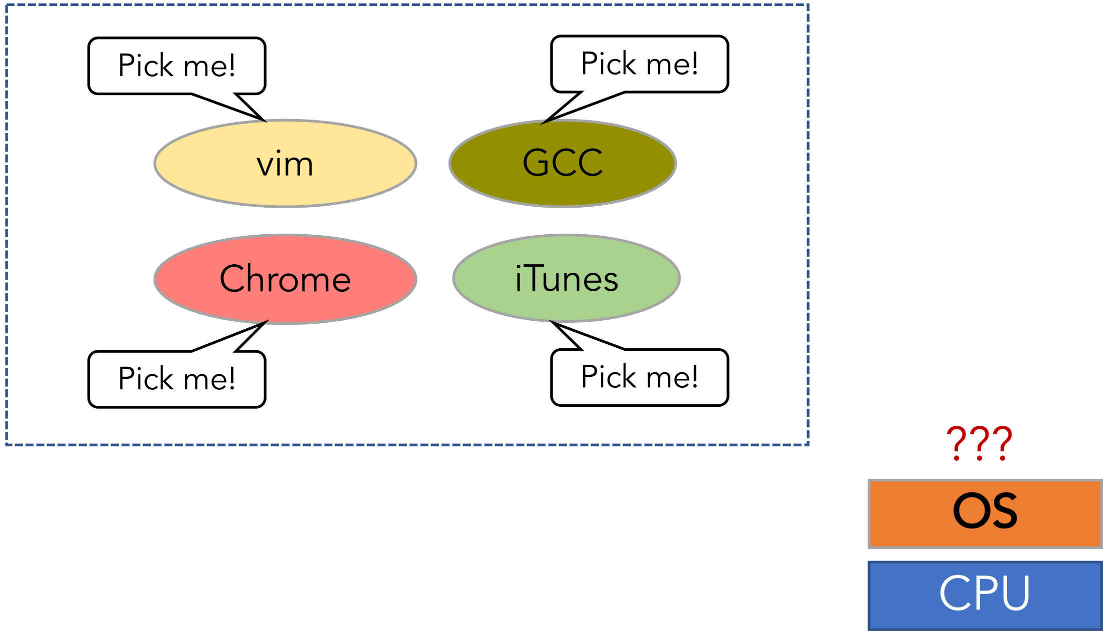
The process is the OS abstraction for CPU (execution)

- It is the unit of execution
- It is the unit of scheduling
- It is the dynamic execution context of a program
- Sometimes also called a job or a task

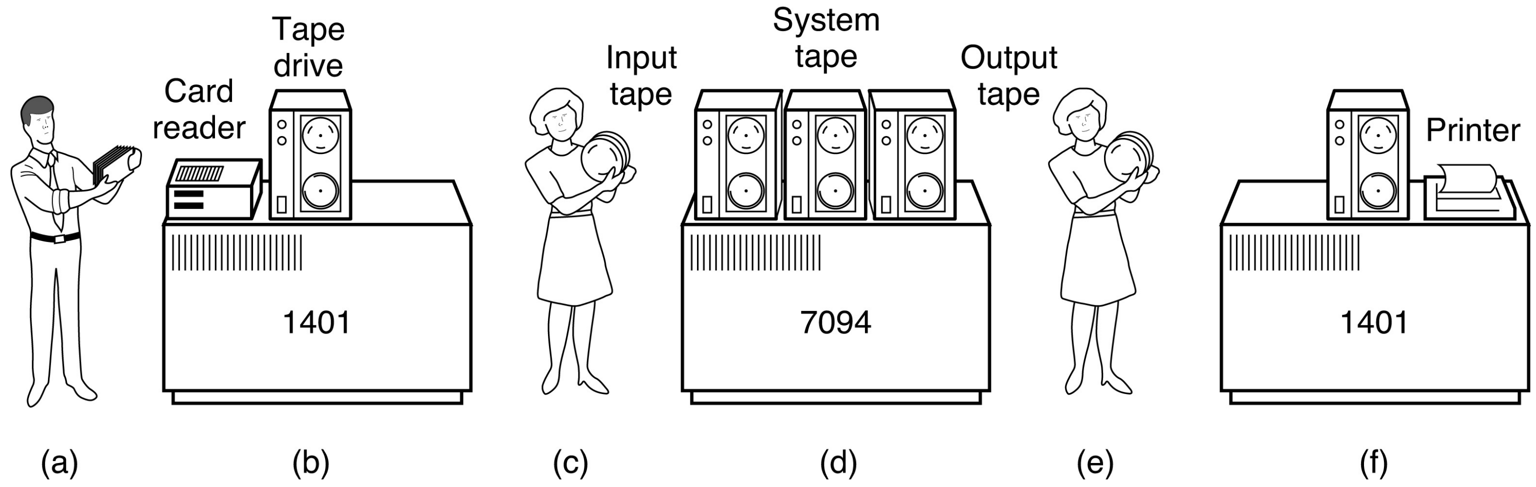
A process is a program in execution

- It defines the sequential, instruction-at-a-time execution of a program
- Programs are static entities with the potential for execution

How Should the OS Manage Processes?



Simple Process Management: One-at-a-time

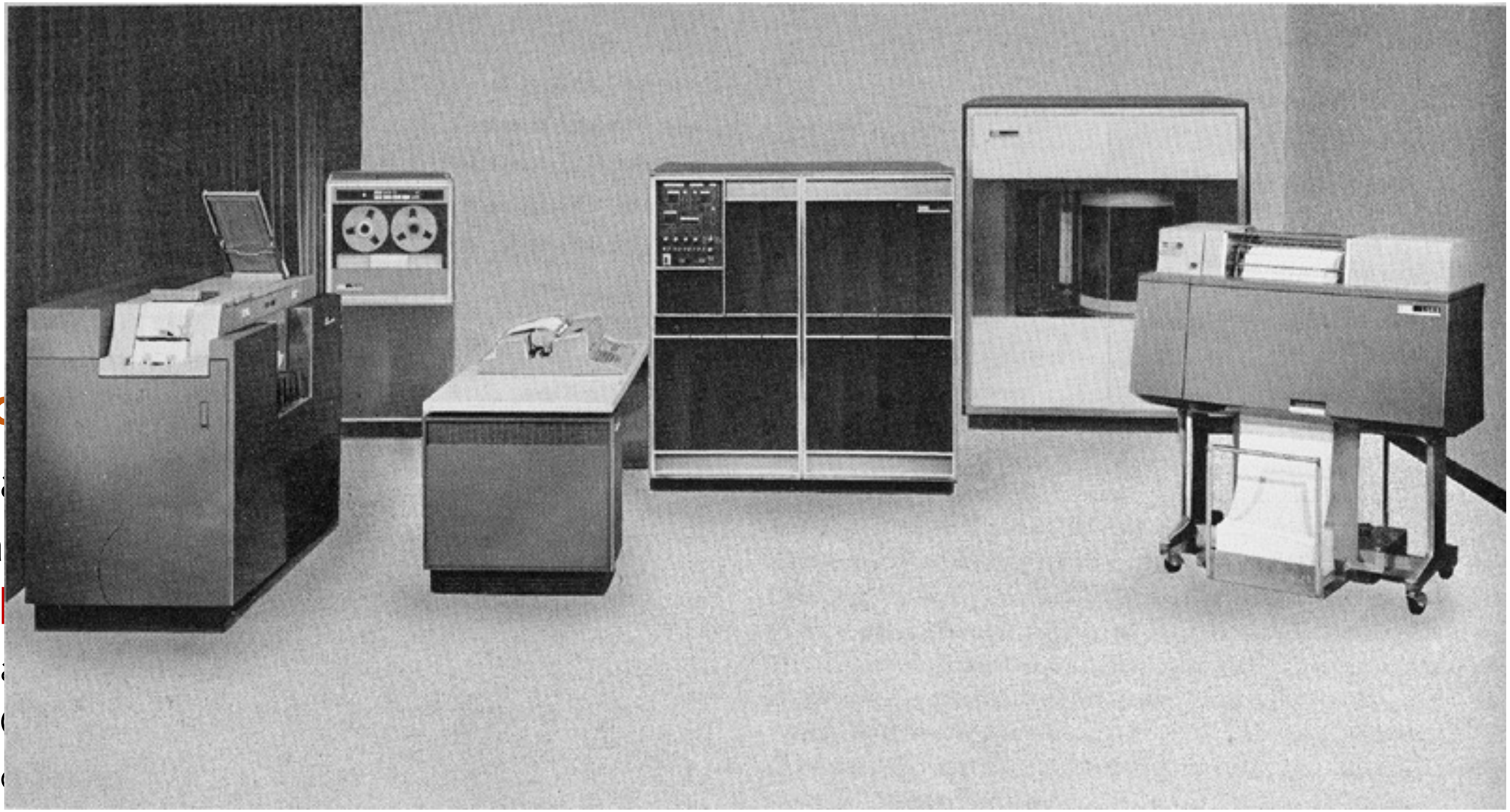


circa 1960s

Uniprogramming: a process runs from start to full completion

- What the early batch operating system does
- Load a job from disk (tape) into memory, execute it, unload the job
- Problem: low utilization of hardware
  - an I/O-intensive process would spend most of its time waiting for punched cards to be read
  - CPU is wasted
  - computers were very expensive back then

Simple Process Management: One-at-a-time



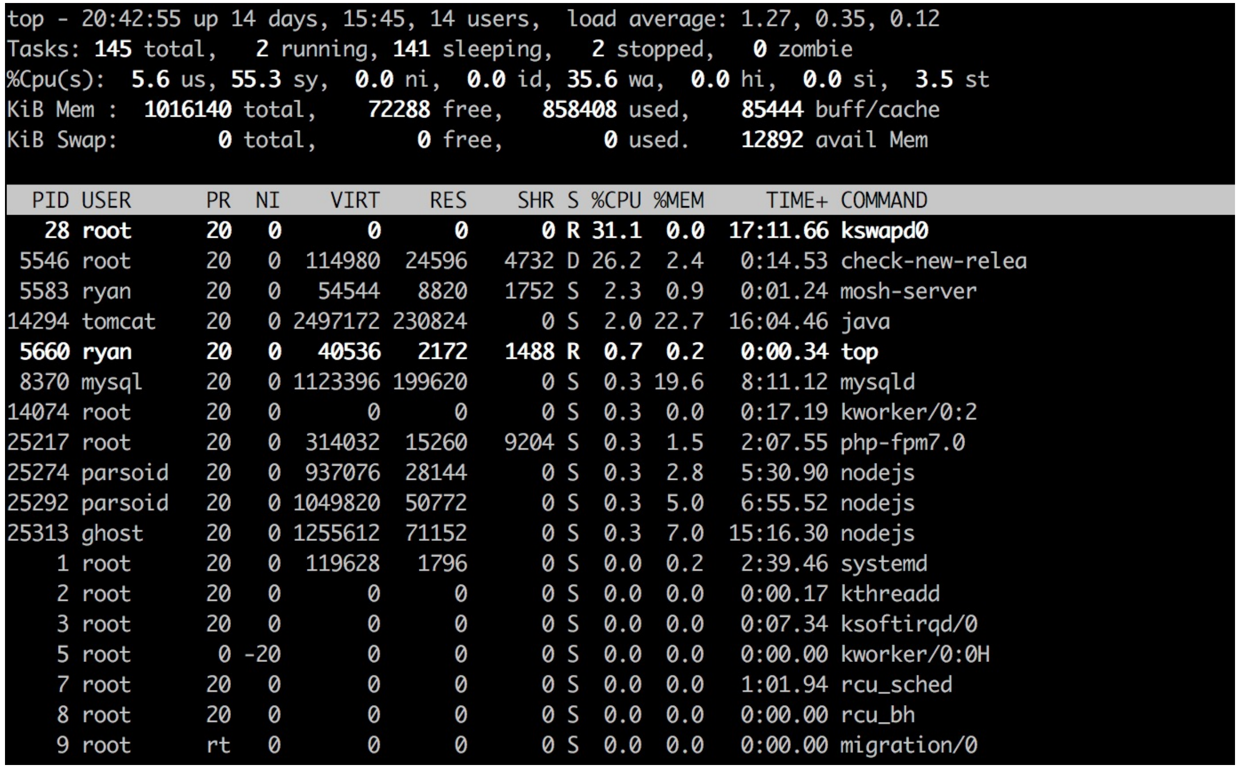
1960s

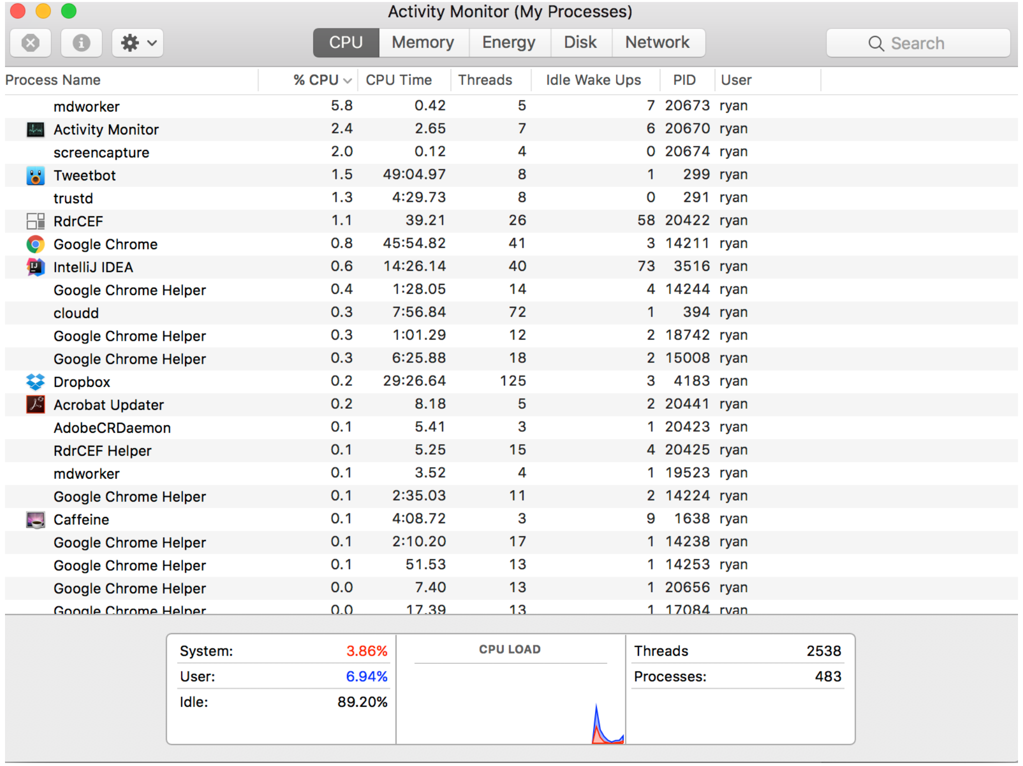
Unipro

- Wh
- Loa
- Prol
  - a ... be read
  - (
  - (

Multiple Processes

Modern OSes run multiple processes simultaneously





Multiple Processes

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
- `vim` – text editor
- `firefox` – web browser

Non-examples (implemented as one process):

- Multiple `firefox` or `tmux` windows (still one process)

Multiprogramming (Multitasking)

Multiprogramming: run more than one process at a time

- Multiple processes loaded in memory and available to run
- If a process is blocked in I/O, select another process to run on CPU
- Different hardware components utilized by different tasks at the same time

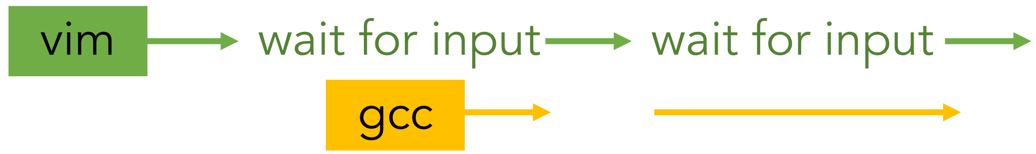
Why multiple processes (multiprogramming)?

- **Advantages:** increase utilization & speed
  - higher throughput
  - lower latency

Increased Utilization & Speed

Multiple processes can increase CPU utilization

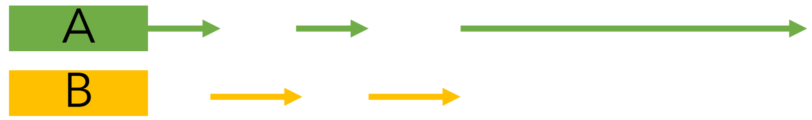
- Overlap one process's computation with another's wait



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

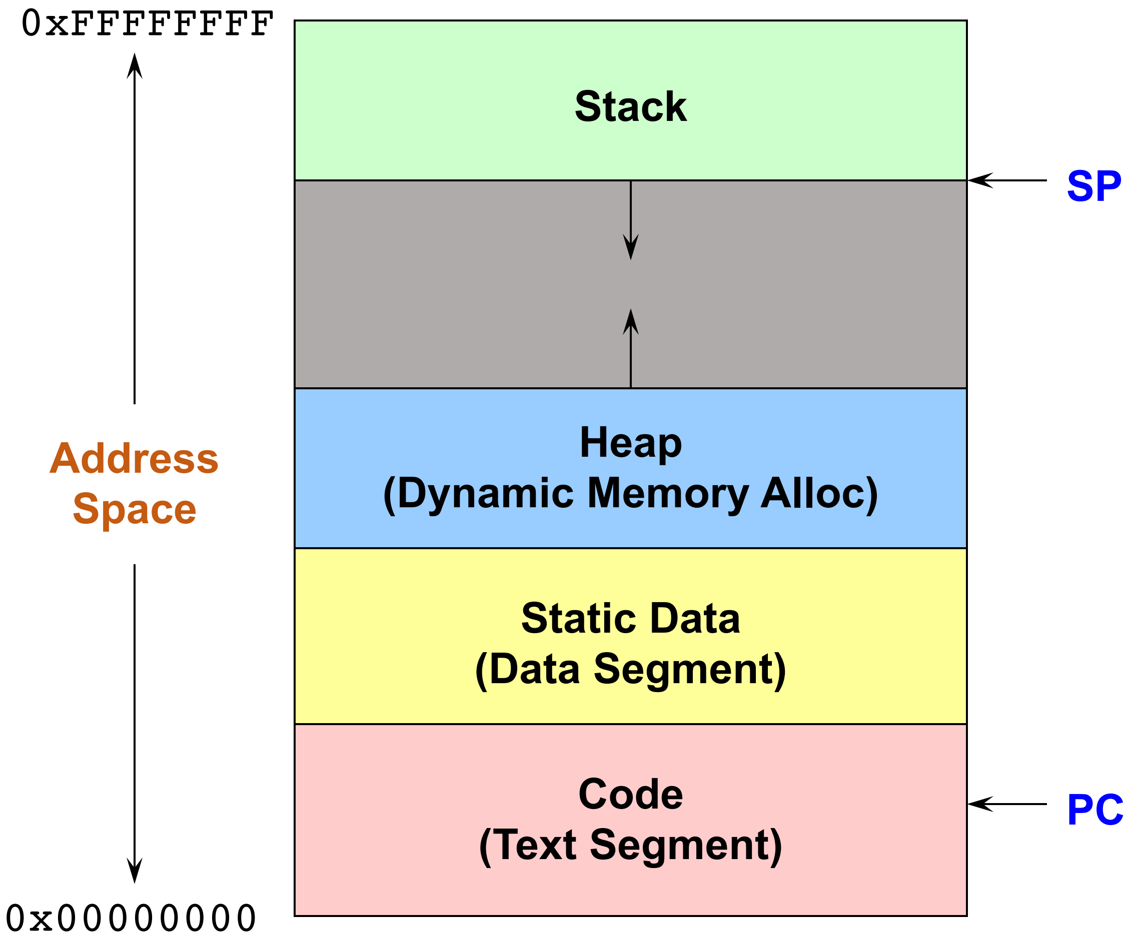
Kernel's View of Processes

Process Components

A process contains all state for a program in execution

- An address space
- The code for the executing program
- The data for the executing program
- An execution stack encapsulating the state of procedure calls
- The program counter (PC) indicating the next instruction
- A set of general-purpose registers with current values
- A set of operating system resources
    - Open files, network connections, etc.

Process Address Space



A Process's View of the World

Each process has own view of machine

- Its own address space
- Its own virtual CPU
- Its own open files

`*(char *)0xc000` **means different thing in P1 & P2**

Simplifies programming model

- `gcc` does not care that `firefox` is running

Naming A Process

A process is named using its process ID (PID)

```
top - 20:42:55 up 14 days, 15:45, 14 users,  load average: 1.27, 0.35, 0.12
Tasks: 145 total,   2 running, 141 sleeping,   2 stopped,   0 zombie
%Cpu(s):  5.6 us, 55.3 sy,  0.0 ni,  0.0 id, 35.6 wa,  0.0 hi,  0.0 si,  3.5 st
KiB Mem :  1016140 total,    72288 free,   858408 used,    85444 buff/cache
KiB Swap:        0 total,        0 free,        0 used.    12892 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
   28 root      20   0       0      0      0 R 31.1  0.0  17:11.66 kswapd0
 5546 root      20   0  114980  24596   4732 D 26.2  2.4   0:14.53 check-new-relea
 5583 ryan      20   0   54544   8820   1752 S  2.3  0.9   0:01.24 mosh-server
14294 tomcat    20   0 2497172 230824      0 S  2.0 22.7  16:04.46 java
 5660 ryan      20   0   40536   2172   1488 R  0.7  0.2   0:00.34 top
 8370 mysql     20   0 1123396 199620      0 S  0.3 19.6   8:11.12 mysqld
14074 root      20   0       0      0      0 S  0.3  0.0   0:17.19 kworker/0:2
25217 root      20   0  314032  15260   9204 S  0.3  1.5   2:07.55 php-fpm7.0
25274 parsoid   20   0  937076  28144      0 S  0.3  2.8   5:30.90 nodejs
25292 parsoid   20   0 1049820  50772      0 S  0.3  5.0   6:55.52 nodejs
25313 ghost     20   0 1255612  71152      0 S  0.3  7.0  15:16.30 nodejs
    1 root      20   0  119628   1796      0 S  0.0  0.2   2:39.46 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.17 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:07.34 ksoftirqd/0
    5 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S  0.0  0.0   1:01.94 rcu_sched
    8 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migration/0
```

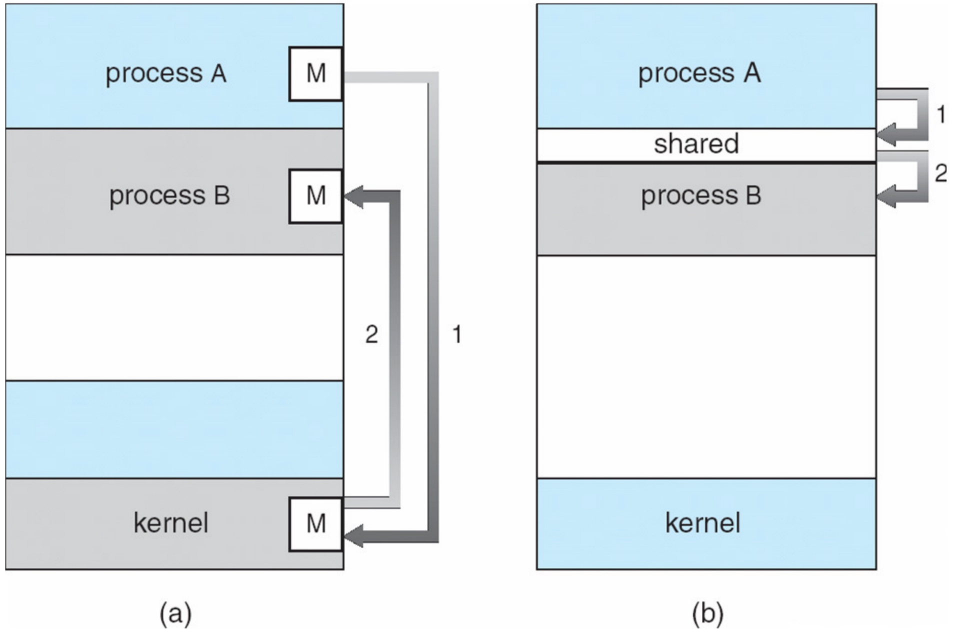
Inter-Process Communication (IPC)

Sometimes want interaction between processes

- Simplest is through files: `vim` edits file, `gcc` compiles it
- More complicated: Shell/command, Window manager/app.

How can processes interact in real time?

Inter-Process Communication (IPC)



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

Implementing Process

A data structure for each process: *Process Control Block* (PCB)

- Contains all the info about a process

Tracks *state* of the process

- Running, ready (runnable), waiting, etc.

PCB includes information necessary for execution

- Registers, virtual memory mappings, open files, etc.
- PCB is also maintained when the process is *not* running (why?)

Various other data about the process

- Credentials (user/group ID), signal mask, priority, accounting, etc.

Process is a heavyweight abstraction!

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

PCB

struct proc (Solaris)

```
/*
 * One structure allocated per active process.  It contains all
 * data needed about the process while the process may be swapped
 * out.  Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 */
typedef struct  proc {
        /*
         * Fields requiring no explicit locking
         */
        struct  vnode *p_exec;          /* pointer to a.out vnode */
        struct  as *p_as;               /* process address space pointer */
        struct  plock *p_lockp;         /* ptr to proc struct's mutex lock */
        kmutex_t p_crlock;              /* lock for p_cred */
        struct  cred    *p_cred;        /* process credentials */
        /*
         * Fields protected by pidlock
         */
        int     p_swapcnt;              /* number of swapped out lwps */
        char    p_stat;                 /* status of process */
        char    p_wcode;                /* current wait code */
        ushort_t p_pidflag;             /* flags protected only by pidlock */
        int     p_wdata;                /* current wait return value */
        pid_t   p_ppid;                 /* process id of parent */
        struct  proc    *p_link;        /* forward link */
        struct  proc    *p_parent;      /* ptr to parent process */
        struct  proc    *p_child;       /* ptr to first child process */
        struct  proc    *p_sibling;     /* ptr to next sibling proc on chain */
        struct  proc    *p_psibling;    /* ptr to prev sibling proc on chain */
        struct  proc    *p_sibling_ns;  /* prt to siblings with new state */
        struct  proc    *p_child_ns;    /* prt to children with new state */
        struct  proc    *p_next;        /* active chain link next */
        struct  proc    *p_prev;        /* active chain link prev */
        struct  proc    *p_nextofkin;   /* gets accounting info at exit */
        struct  proc    *p_orphan;
        struct  proc    *p_nextorph;
```

```
*p_pglink;      /* process group hash chain link next */
struct  proc    *p_ppglink;     /* process group hash chain link prev */
struct  sess    *p_sessp;       /* session information */
struct  pid     *p_pidp;        /* process ID info */
struct  pid     *p_pgidp;       /* process group ID info */
/*
 * Fields protected by p_lock
 */
kcondvar_t p_cv;                /* proc struct's condition variable */
kcondvar_t p_flag_cv;
kcondvar_t p_lwpexit;           /* waiting for some lwp to exit */
kcondvar_t p_holdlwps;          /* process is waiting for its lwps */
                                /* to to be held.  */
ushort_t p_pad1;                /* unused */
uint_t  p_flag;                 /* protected while set. */

/* flags defined below */
clock_t p_utime;                /* user time, this process */
clock_t p_stime;                /* system time, this process */
clock_t p_cutime;               /* sum of children's user time */
clock_t p_cstime;               /* sum of children's system time */
caddr_t *p_segacct;             /* segment accounting info */
caddr_t p_brkbase;              /* base address of heap */
size_t  p_brksize;              /* heap size in bytes */
/*
 * Per process signal stuff.
 */
k_sigset_t p_sig;               /* signals pending to this process */
k_sigset_t p_ignore;            /* ignore when generated */
k_sigset_t p_siginfo;           /* gets signal info with signal */
struct sigqueue *p_sigqueue;    /* queued siginfo structures */
struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
struct sigqhdr *p_signhdr;      /* hdr to signotify structure pool */
uchar_t p_stopsig;              /* jobcontrol stop signal */
```

struct proc (Solaris) (2)

```
/*
 * Special per-process flag when set will fix misaligned memory
 * references.
 */
char    p_fixalignment;

/*
 * Per process lwp and kernel thread stuff
 */
id_t    p_lwpid;                /* most recently allocated lwpid */
int     p_lwpcnt;               /* number of lwps in this process */
int     p_lwprcnt;              /* number of not stopped lwps */
int     p_lwpwait;              /* number of lwps in lwp_wait() */
int     p_zombcnt;              /* number of zombie lwps */
int     p_zomb_max;             /* number of entries in p_zomb_tid */
id_t    *p_zomb_tid;            /* array of zombie lwpids */
kthread_t *p_tlist;             /* circular list of threads */
/*
 * /proc (process filesystem) debugger interface stuff.
 */
k_sigset_t p_sigmask;           /* mask of traced signals (/proc) */
k_fltset_t p_fltmask;           /* mask of traced faults (/proc) */
struct  vnode *p_trace;         /* pointer to primary /proc vnode */
struct  vnode *p_plist;         /* list of /proc vnodes for process */
kthread_t *p_agenttp;           /* thread ptr for /proc agent lwp */
struct watched_area *p_warea;   /* list of watched areas */
ulong_t p_nwarea;               /* number of watched areas */
struct watched_page *p_wpage;   /* remembered watched pages (vfork) */
int     p_nwpage;               /* number of watched pages (vfork) */
int     p_mapcnt;               /* number of active pr_mappage()s */
struct  proc  *p_rlink;         /* linked list for server */
kcondvar_t p_srwchan_cv;
size_t  p_stksize;              /* process stack size in bytes */
```

```
/*
 * Microstate accounting, resource usage, and real-time profiling
 */
hrtime_t p_mstart;              /* hi-res process start time */
hrtime_t p_mterm;               /* hi-res process termination time */
hrtime_t p_mlreal;              /* elapsed time sum over defunct lwps */
hrtime_t p_acct[NMSTATES];      /* microstate sum over defunct lwps */
struct lrusage p_ru;            /* lrusage sum over defunct lwps */
struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
uintptr_t p_rprof_cyclic;       /* ITIMER_REALPROF cyclic */
uint_t  p_defunct;              /* number of defunct lwps */
/*
 * profiling. A lock is used in the event of multiple lwp's
 * using the same profiling base/size.
 */
kmutex_t p_pflock;              /* protects user profile arguments */
struct prof p_prof;             /* profile arguments */

/*
 * The user structure
 */
struct user p_user;             /* (see sys/user.h) */

/*
 * Doors.
 */
kthread_t               *p_server_threads;
struct door_node        *p_door_list;   /* active doors */
struct door_node        *p_unref_list;
kcondvar_t              p_server_cv;
char                    p_unref_thread; /* unref thread created */
```

`struct proc` (Solaris) (3)

```
        /*
         * Kernel probes
         */
        uchar_t                 p_tnf_flags;

        /*
         * C2 Security  (C2_AUDIT)
         */
        caddr_t p_audit_data;           /* per process audit structure */
        kthread_t       *p_aslwptp;     /* thread ptr representing "aslwp" */
#if defined(i386) || defined(__i386) || defined(__ia64)
        /*
         * LDT support.
         */
        kmutex_t p_ldtlock;             /* protects the following fields */
        struct seg_desc *p_ldt;         /* Pointer to private LDT */
        struct seg_desc p_ldt_desc;     /* segment descriptor for private LDT */
        int p_ldtlimit;                 /* highest selector used */
#endif
        size_t p_swrss;                 /* resident set size before last swap */
        struct aio      *p_aio;         /* pointer to async I/O struct */
        struct itimer   **p_itimer;     /* interval timers */
        k_sigset_t      p_notifsigs;    /* signals in notification set */
        kcondvar_t      p_notifcv;      /* notif cv to synchronize with aslwp */
        timeout_id_t    p_alarmid;      /* alarm's timeout id */
        uint_t          p_sc_unblocked; /* number of unblocked threads */
        struct vnode    *p_sc_door;     /* scheduler activations door */
        caddr_t         p_usrstack;     /* top of the process stack */
        uint_t          p_stkprot;      /* stack memory protection */
        model_t         p_model;        /* data model determined at exec time */
        struct lwpchan_data     *p_lcp; /* lwpchan cache */

        /*
         * protects unmapping and initilization of robust locks.
         */
        kmutex_t        p_lcp_mutexinitlock;
        utrap_handler_t *p_utraps;      /* pointer to user trap handlers */
        refstr_t        *p_corefile;    /* pattern for core file */

#if defined(__ia64)
        caddr_t         p_upstack;      /* base of the upward-growing stack */
        size_t          p_upstksize;    /* size of that stack, in bytes */
        uchar_t         p_isa;          /* which instruction set is utilized */
#endif
        void            *p_rce;         /* resource control extension data */
        struct task     *p_task;        /* our containing task */
        struct proc     *p_taskprev;    /* ptr to previous process in task */
        struct proc     *p_tasknext;    /* ptr to next process in task */
        int             p_lwpdaemon;    /* number of TP_DAEMON lwps */
        int             p_lwpdwait;     /* number of daemons in lwp_wait() */
        kthread_t       **p_tidhash;    /* tid (lwpid) lookup hash table */
        struct sc_data  *p_schedctl;    /* available schedctl structures */
} proc_t;
```

Process State

A process has an execution state to indicate what it is doing

Running: Executing instructions on the CPU

- It is the process that has control of the CPU
- How many processes can be in the running state simultaneously?

Ready (runnable): Waiting to be assigned to the CPU

- Ready to execute, but another process is executing on the CPU

Waiting: Waiting for an event, e.g., I/O completion

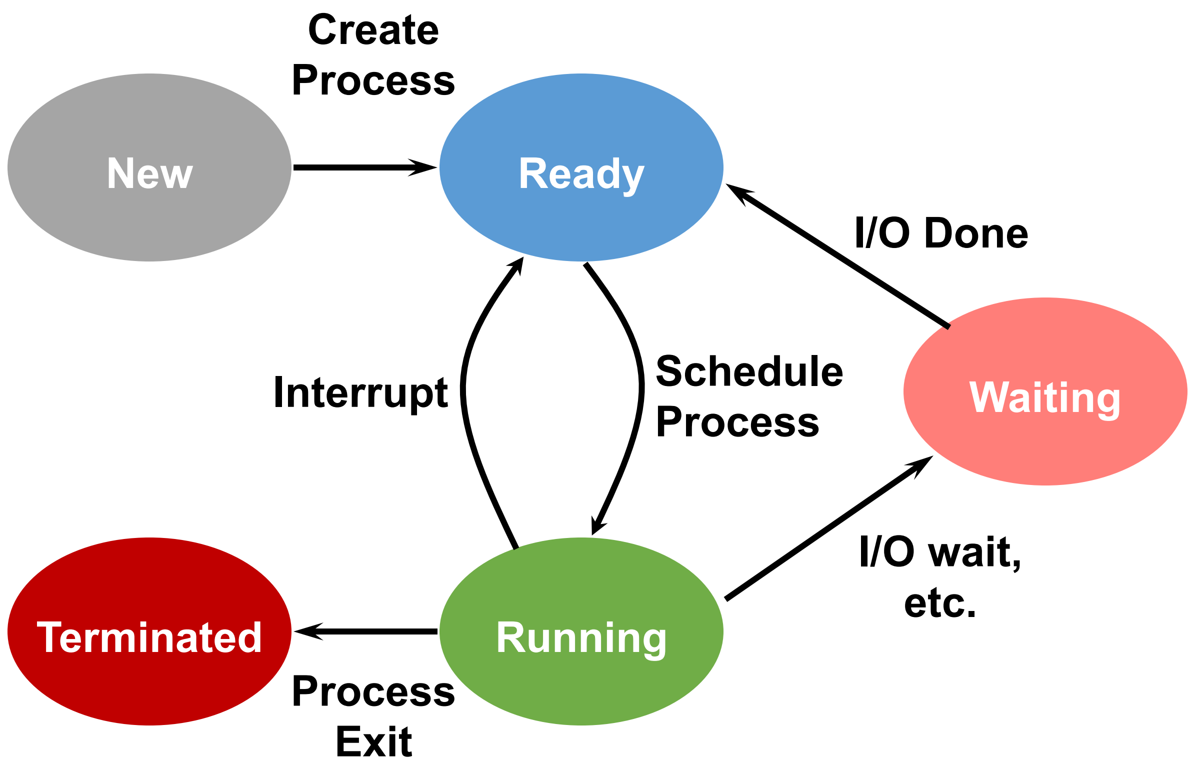
- It cannot make progress until event is signaled (disk completes)

Transition of Process State

As a process executes, it moves from state to state

- Unix `ps`: `STAT` column indicates execution state
- What state do you think a process is in most of the time?
- How many processes can a system support?

Process State Graph



State Queues

How does the OS keep track of processes?

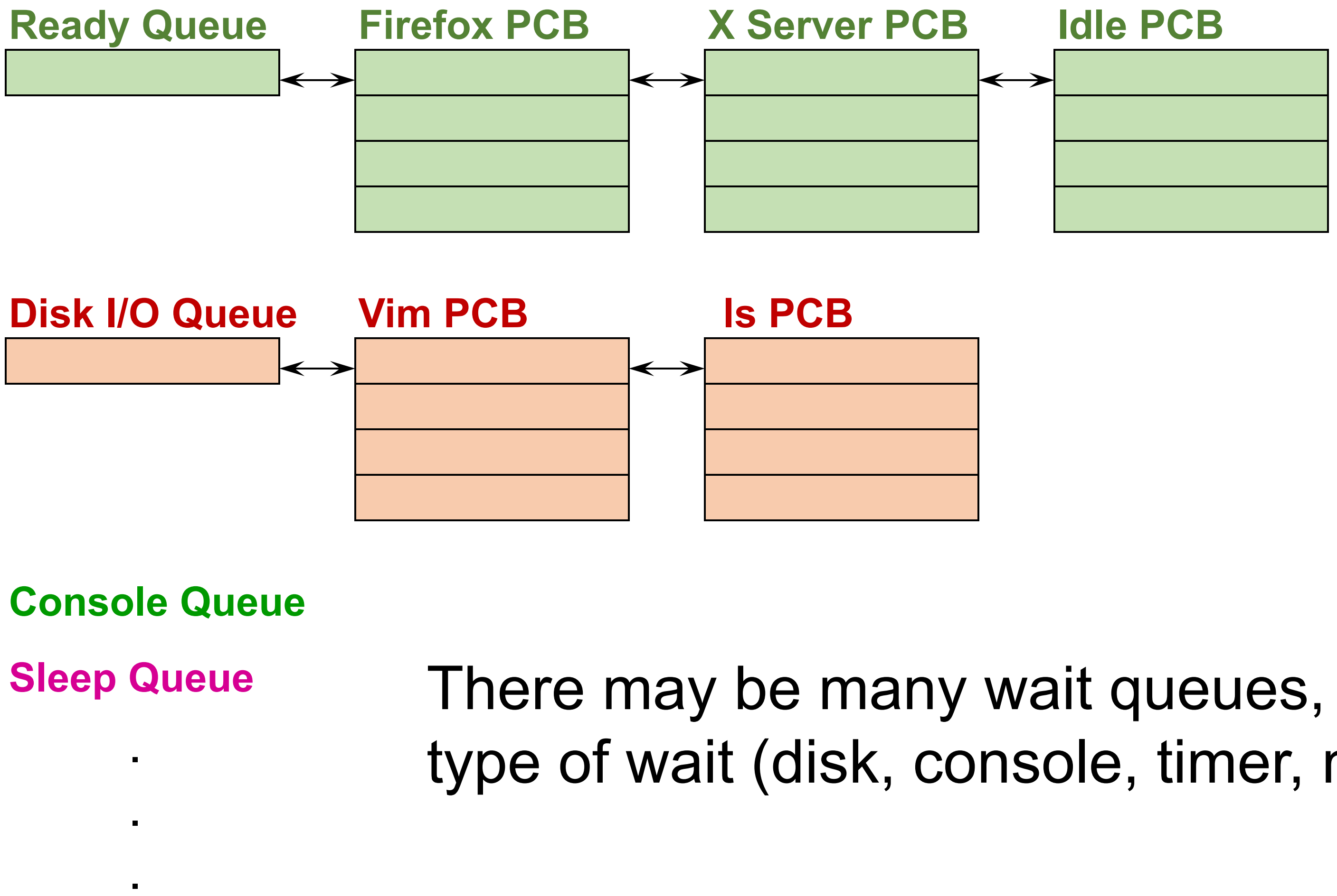
Naïve approach: process list

- How to find out processes in the ready state?
  - Iterate through the list
- Problem: slow!

Improvement: partition list based on states

- OS maintains a collection of queues that represent the state of all processes
- Typically, one queue for each state: ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is moved from one queue into another

State Queues



There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

Questions?

Scheduling

Which process should kernel run?

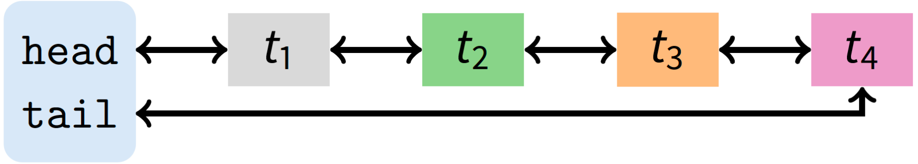
- if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
- if >1 runnable, must make scheduling decision

Scan process table for first runnable?

- Expensive. Unfairness (small pids do better)

FIFO?

- Put tasks on back of list, pull them from front:
- Pintos does this—see `ready_list` in `thread.c`



Priority?

Discuss in later lecture in detail

Preemption

When to trigger a process scheduling decision?

- Yield control of CPU
  - voluntarily, e.g., `sched_yield`
  - system call, page fault, illegal instruction, etc.
- Preemption

Periodic timer interrupt

- If running process used up quantum, schedule another

Device interrupt

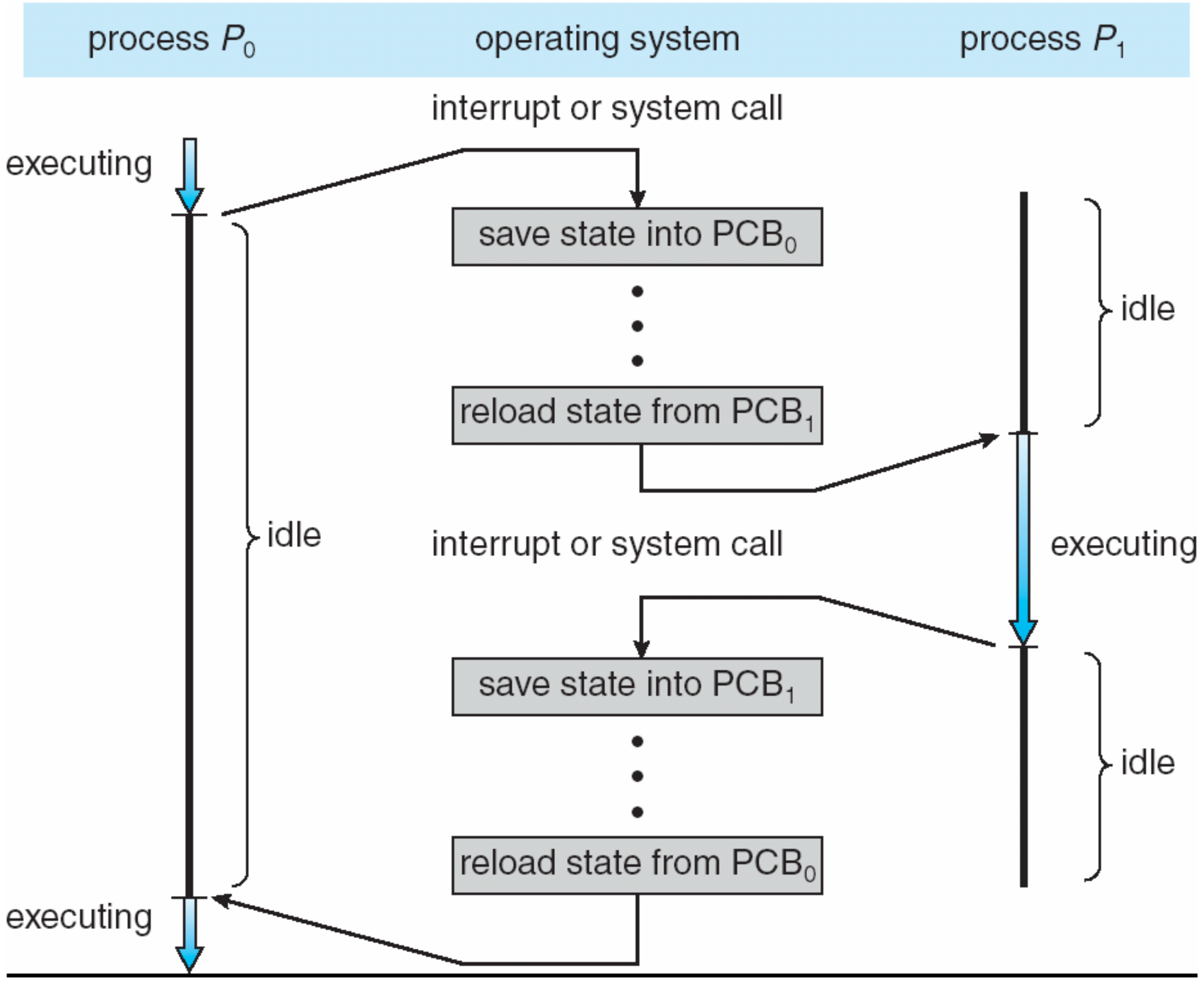
- Disk request completed, or packet arrived on network
- Previously waiting process becomes runnable

Preemption → Context Switch

Changing running process is called a context switch

- CPU hardware state is changed from one to another
- This can happen 100 or 1000 times a second!

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)

Usually causes more cache misses (switch working sets)

Questions?

User's (Programmer's) View of Processes

Process-Related System Calls

Allow a program to create a child process

Creating a Process

A process is created by another process

- Parent is creator, child is created (Unix: ps "PPID" field)
- What creates the first process (Unix: init (PID 0 or 1))?

Parent defines resources and privileges for its children

- Unix: Process User ID is inherited – children of your shell execute with your privileges

After creating a child

- the parent may either wait for it to finish its task or continue in parallel

Process Creation: Windows

The system call on Windows for creating a process is called, surprisingly enough, `CreateProcess`:

`BOOL CreateProcess(char *prog, char *args)` (simplified)

`CreateProcess`

1. Creates and initializes a new PCB
2. Creates and initializes a new address space
3. Loads the program specified by "`prog`" into the address space
4. Copies "`args`" into memory allocated in address space
5. Initializes the saved hardware context to start execution at main (or as specified)
6. Places the PCB on the ready queue

Microsoft | Windows Dev Center

Sign in

Home | Explore | Docs | Downloads | Samples | Community | Programs | Dashboard

Windows desktop applications > Develop > Desktop technologies > System Services > Processes and Threads > Process and Thread Reference > Process and Thread Functions > CreateProcess

CreateProcess function

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

Syntax

C++

```
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
);
```

Process Creation: Unix

In Unix, processes are created using fork()

```
int fork()
```

`fork()`

1. Creates and initializes a new PCB
2. Creates a new address space
3. Initializes the address space with a **copy** of the address space of the parent
4. Initializes the kernel resources to point to the parent's resources (e.g., open files)
5. Places the PCB on the ready queue

Fork returns twice

- Huh?
- Returns the child's PID to the parent, "0" to the child

Mac Developer Library

OS X Man Pages

FORK(2) BSD System Calls Manual FORK(2)

NAME

fork -- create a new process

SYNOPSIS

#include <unistd.h>

pid_t
fork(void);

DESCRIPTION

Fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES

Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

ERRORS

Fork() will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.

[EAGAIN] The system-imposed limit MAXUPRC (<sys/param.h>) on the total number of processes under execution by a single user would be exceeded.

fork()

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s is %d\n", name, getpid());
    return 0;
  } else {
    printf("My child is %d\n", child_pid);
    return 0;
  }
}
```

What does this program print?

Example Output

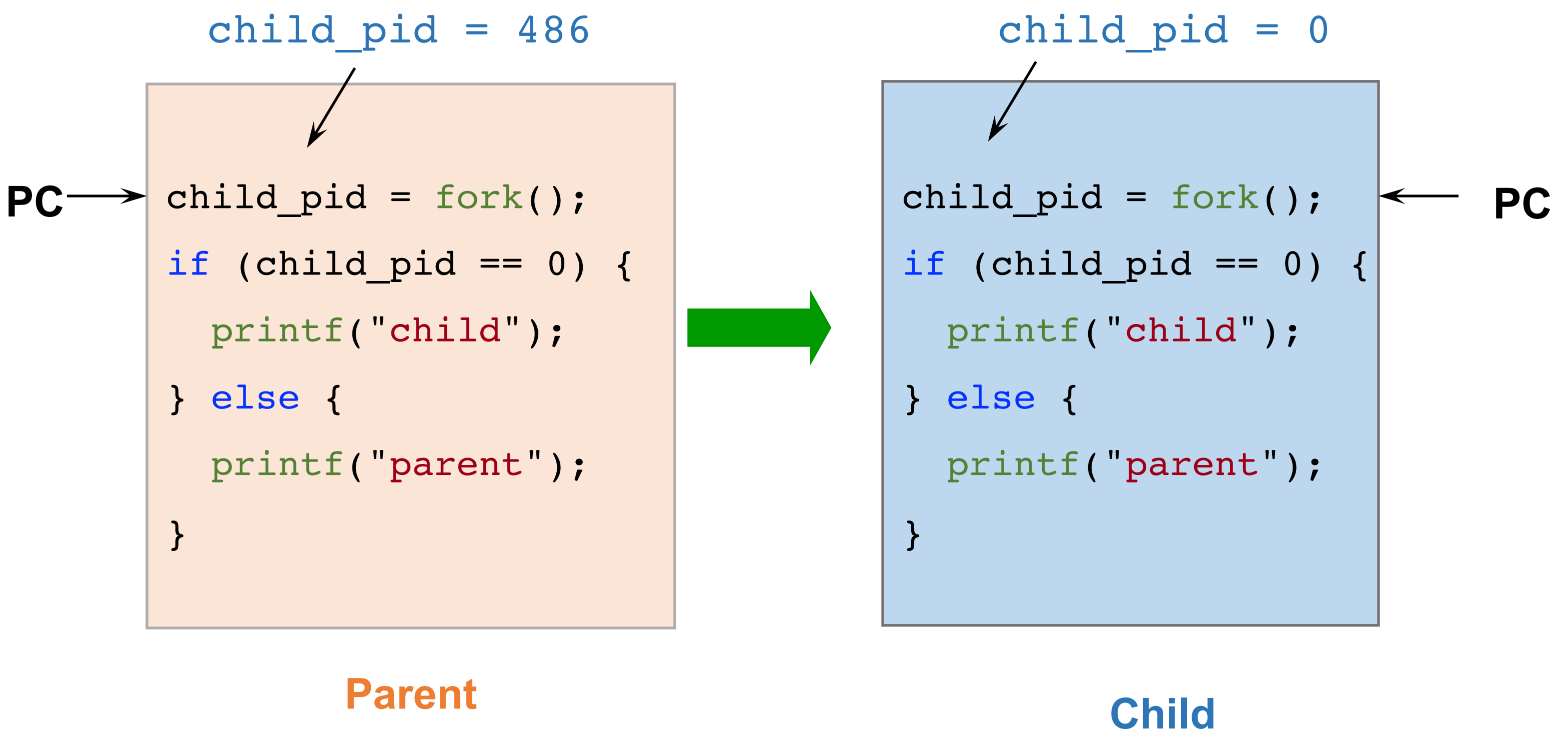
```
$ gcc -o fork fork.c

$ ./fork
```

My child is 486

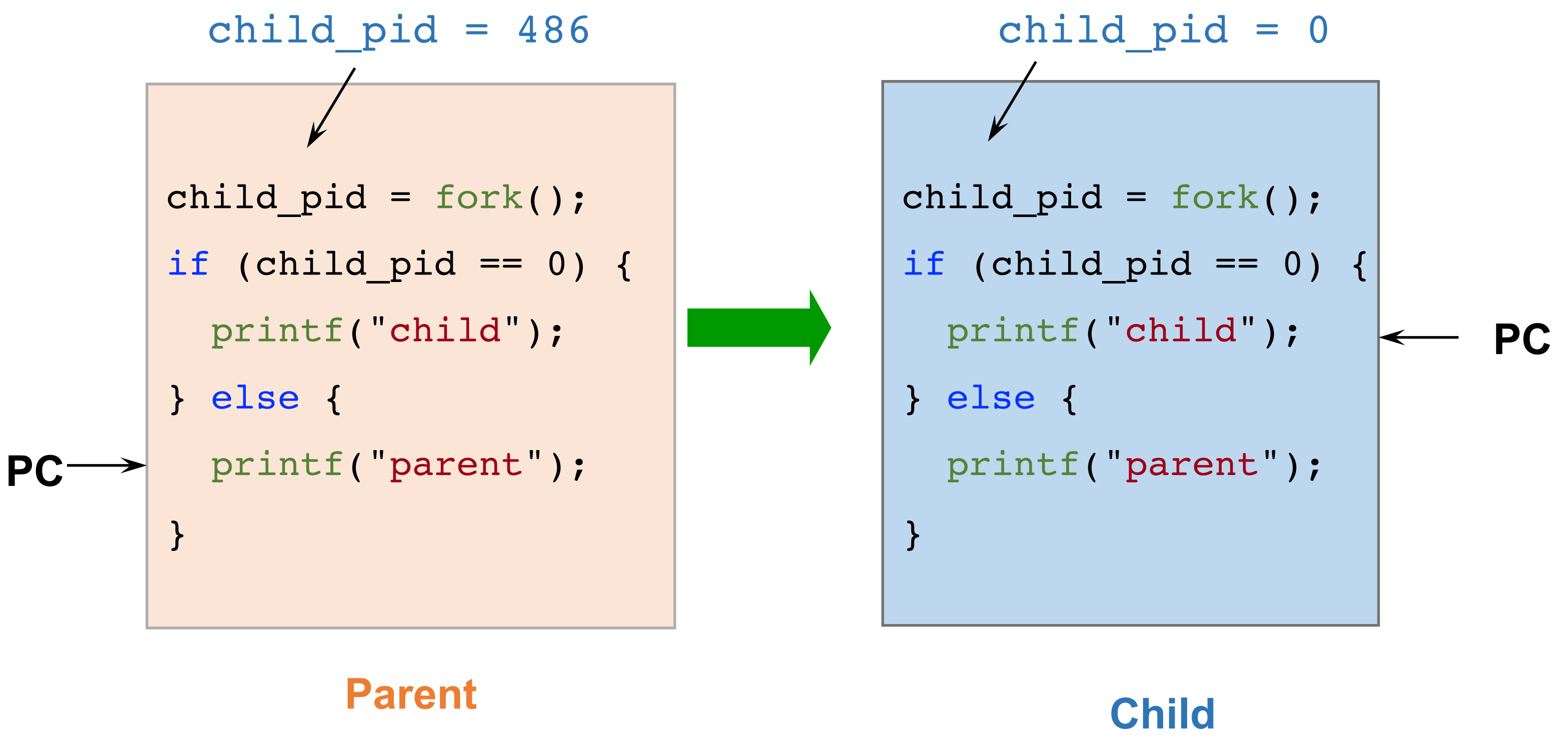
Child of ./fork is 486

Duplicating Address Spaces



The hardware contexts stored in the PCBs of the two processes will be identical, meaning the `EIP` register will point to the same instruction

Divergence


child_pid = 486
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Parent
child_pid = 0
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Child

Example Continued

```
$ gcc -o fork fork.c

$ ./fork
```

My child is 486

Child of ./fork is 486

```
$ ./fork
```

Child of ./fork is 498

My child is 498

Why is the output in a different order?

Process Creation: Unix (2)

Wait a second. How do we actually start a new program?

```
int execv(char *prog, char *argv[])
int execve(const char *filename, char *const argv[], char *const envp[])
```

`execv()`

1. Stops the current process
2. Loads the program "prog" into the process' address space
3. Initializes hardware context and args for the new program
4. Places the PCB onto the ready queue

- Note: It **does not** create a new process

What does it mean for `exec` to return?

Warning: Pintos exec more like combined fork/exec

Why `fork()`?

Most calls to `fork` followed by `exec`

- could also combine into one `spawn` system call

Very useful when the child…

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

Example: web server

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    // Handle client request
  } else {
    // Close socket
  }
}
```

Why `fork()`?

Most calls to `fork` followed by `exec`

- could also combine into one `spawn` system call

Very useful when the child…

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

Example: web server

Example: shell

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;
  }
}
```

https://www.cs.jhu.edu/~huang/cs318/fall22/code/minish.c

```
~/318 $ gcc -o minish minish.c
~/318 $ ./minish
$ date
Wed Aug 26 08:39:26 EDT 2021
$ /usr/bin/vim --version
VIM - Vi IMproved 8.1 (2018 May
18, compiled Jun  5 2020 21:30:37)
macOS version
…
```

Why `fork()`?

Most calls to `fork` followed by `exec`

- could also combine into one `spawn` system call

Very useful when the child…

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

Real win is simplicity of interface

- Tons of things you might want to do to child:
  - manipulate file descriptors, set environment variables, reduce privileges, ...
- **Yet `fork` requires no arguments at all**

redirsh.c

Shell command redirection: `command < input > output 2> errlog`

https://www.cs.jhu.edu/~huang/cs318/fall22/code/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);
}
```

```
~/318 $ gcc -o redirsh redirsh.c
~/318 $ ./redirsh
$ ls > list.txt
$ sort < list.txt > sorted_list.txt
$ cat sorted_list.txt
a.c
b.c
cs318.txt
…
```

Spawning a Process Without `fork`

Without fork, needs tons of different options for new process

- 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
);
```

Process Creation: Unix (3)

Why Windows use `CreateProcess` while Unix uses `fork`/`exec`?

- different OS design philosophy

What happens if you run "exec csh" in your shell?

What happens if you run "exec ls" in your shell? Try it.

`fork()` **can return an error. Why might this happen?**

Process Termination

All good processes must come to an end. But how?

- Unix: `exit(int status)`, Windows: `ExitProcess(int status)`

Essentially, free resources and terminate

1. Terminate all threads (next lecture)
2. Close open files, network connections
3. Allocated memory (and VM pages out on disk)
4. Remove PCB from kernel data structures, delete

Note that a process does *not* need to clean up itself

- Why does the OS have to do it?

`wait()` a second…

Often it is convenient to pause until a child process has finished

- Think of executing commands in a shell

Unix `wait(int *wstatus)` (Windows: `WaitForSingleObject`)

- Suspends the current process until *any* child process ends
- `waitpid()` suspends until the specified child process ends

`wait()` has a return value…what is it?

Unix: Every process must be "reaped" by a parent

- What happens if a parent process exits before a child?
- What do you think a "zombie" process is?

Process Summary

What are the units of execution?
- Processes

How are those units of execution represented?
- Process Control Blocks (PCBs)

How is work scheduled in the CPU?
- Process states, process queues, context switches

What are the possible execution states of a process?
- Running, ready, waiting

How does a process move from one state to another?
- Scheduling, I/O, creation, termination

How are processes created?
- `CreateProcess` (NT), `fork/exec` (Unix)

Next time…

Read Chapters 26, 27

Lab 0 due

Lab 1 starts