CSE120
Principles of Operating Systems

Prof Yuanyuan (YY) Zhou

Process*

Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in CSE120
  - You can rest assured that they will be covered in the exams
- Today's topics are processes and process management

Users, Programs

- Users have accounts on the system
- Users launch programs
  - Many users may launch same program
  - One user may launch many instances of the same program
- What programs have you launched in your phones, laptops, ipads?
- Then what is a process?

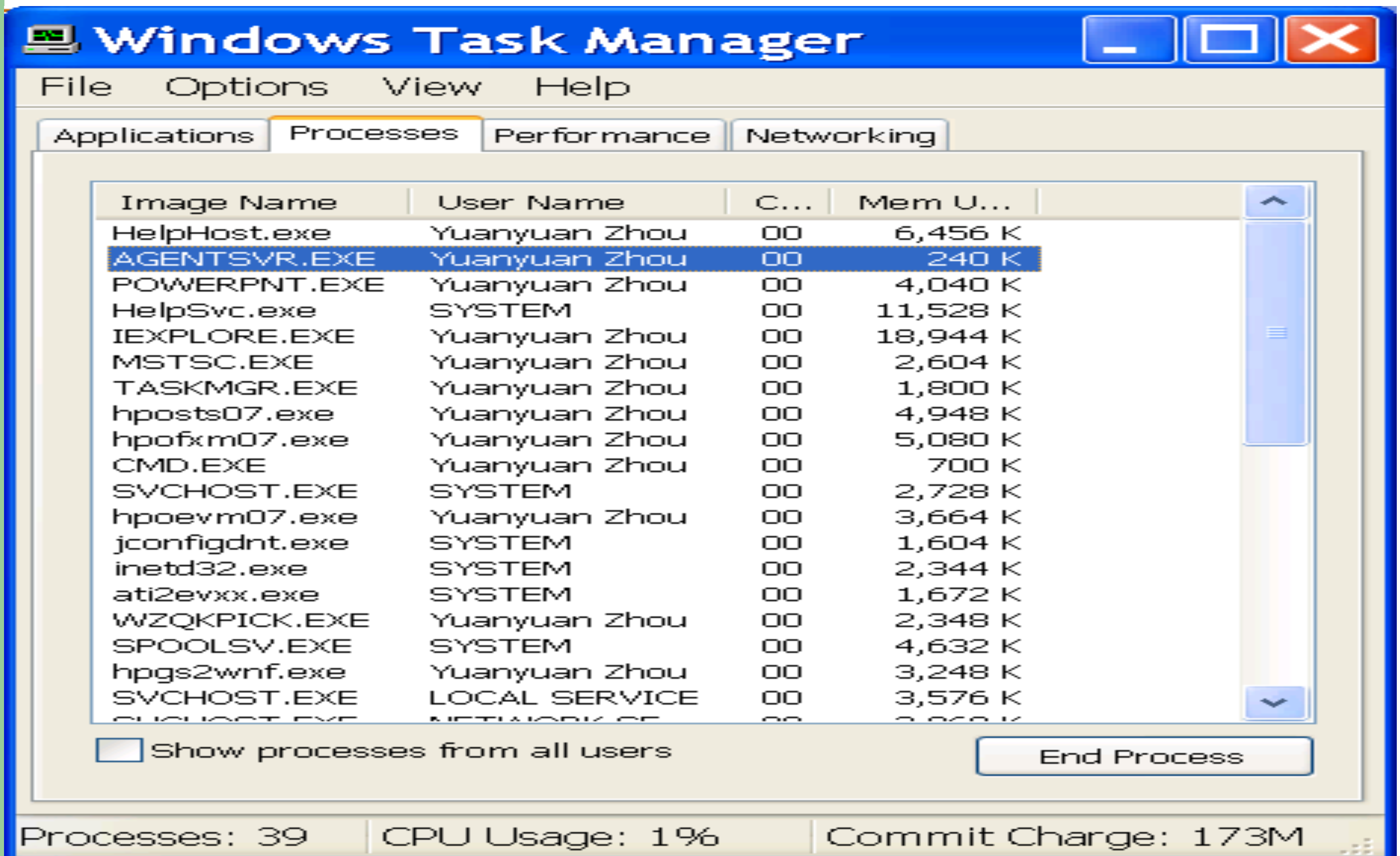
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the dynamic execution context of a program
- A process is sometimes called a job or a task
- Real life analogy?

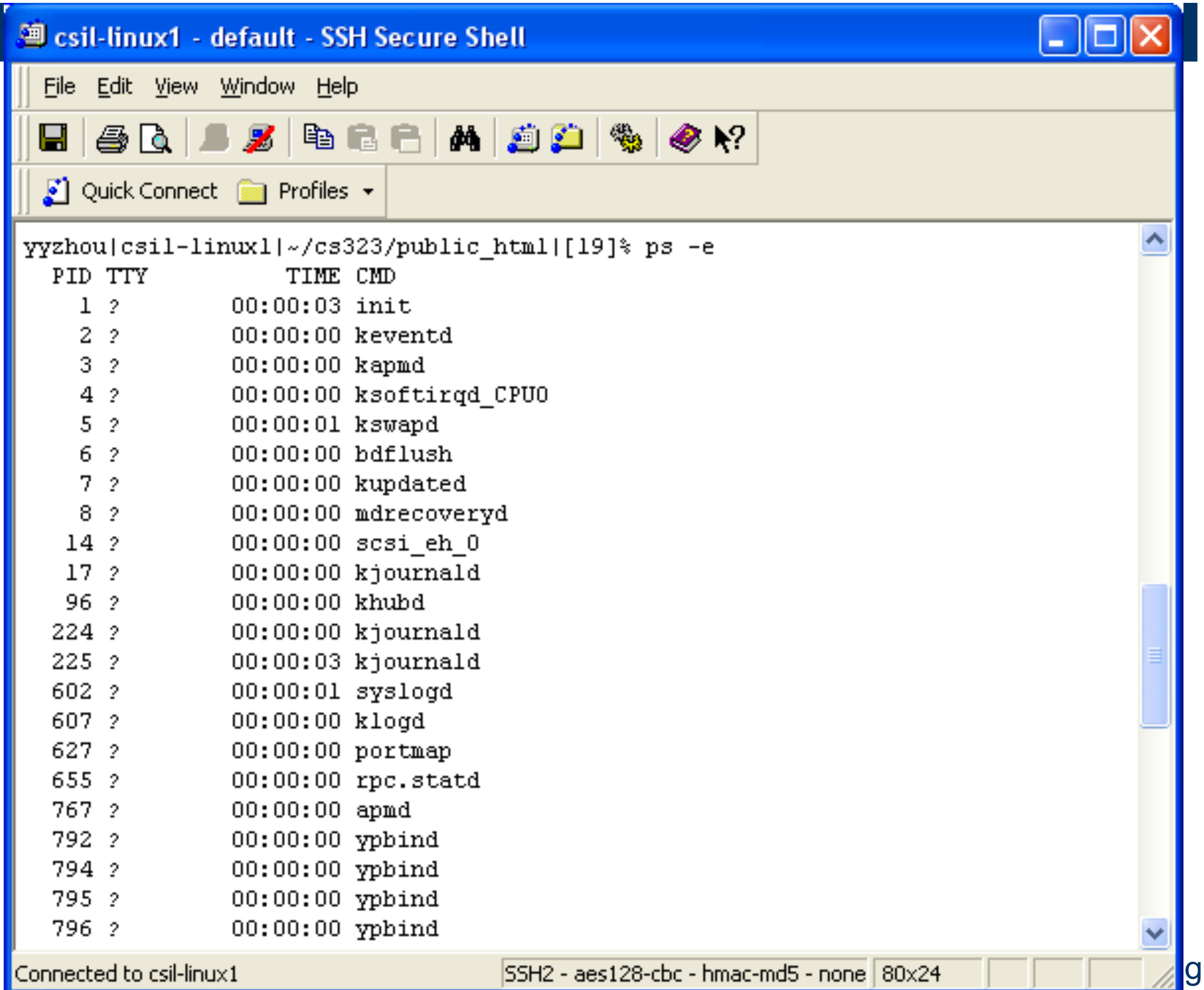
Analogy: A robot taking CSE120

- Program: steps for attending the lecture
  - Step1: walk to Center
  - Step2: enter 109
  - Step3: find a seat
  - Step4: listen and take notes (or sleep)
- Process: attending the lecture
  - Action
  - Each of you now are in the middle of a process

Windows Task Manager



Linux Example: ps



Mac OS – Activity Monitor

Activity Monitor

Quit Process | Inspect | Sample Process

Show: My Processes | Filter

PID	Process Name	User	% CPU	Threads	Real Mem	Kind
237	Dock	yyzhou	6.1	8	16.0 MB	Intel (64 bit)
82122	RealPlayer Downloader	yyzhou	1.3	10	18.3 MB	Intel
947	Safari Web Content	yyzhou	1.1	21	1.09 GB	Intel (64 bit)
89666	screencapture	yyzhou	1.0	2	3.9 MB	Intel (64 bit)
89661	Activity Monitor	yyzhou	0.9	2	36.4 MB	Intel (64 bit)
80846	Microsoft PowerPoint	yyzhou	0.8	14	140.1 MB	Intel
81024	Microsoft Word	yyzhou	0.7	6	76.9 MB	Intel
89034	Google Chrome Helper	yyzhou	0.7	13	67.1 MB	Intel (64 bit)
277	RealPlayer Downloader Agent	yyzhou	0.6	4	1.8 MB	Intel
82746	Microsoft Excel	yyzhou	0.6	10	13.4 MB	Intel
80525	Adobe Reader and Acrobat Updater	yyzhou	0.5	8	4.3 MB	Intel (64 bit)
275	Dropbox	yyzhou	0.4	93	60.4 MB	Intel (64 bit)
276	Fitbit Connect	yyzhou	0.4	6	6.5 MB	Intel (64 bit)
86808	GoogleTalkPlugin	yyzhou	0.3	12	2.1 MB	Intel
88591	Google Chrome Helper	yyzhou	0.2	15	24.4 MB	Intel (64 bit)
89367	Google Chrome Helper	yyzhou	0.2	9	63.1 MB	Intel (64 bit)
88586	Google Chrome Helper	yyzhou	0.1	17	255.7 MB	Intel (64 bit)
89032	Google Chrome Helper	yyzhou	0.1	12	46.2 MB	Intel (64 bit)
88569	Google Chrome	yyzhou	0.1	48	134.3 MB	Intel (64 bit)
89131	Google Chrome Helper	yyzhou	0.1	9	27.0 MB	Intel (64 bit)
278	IDriveMonitor	yyzhou	0.0	6	8.8 MB	Intel
945	Safari	yyzhou	0.0	19	152.2 MB	Intel (64 bit)
270	Start SmartMobility Services	yyzhou	0.0	11	2.1 MB	Intel (64 bit)

CPU | System Memory | Disk Activity | Disk Usage | Network

% User: 8.00
% System: 7.88
% Idle: 84.12

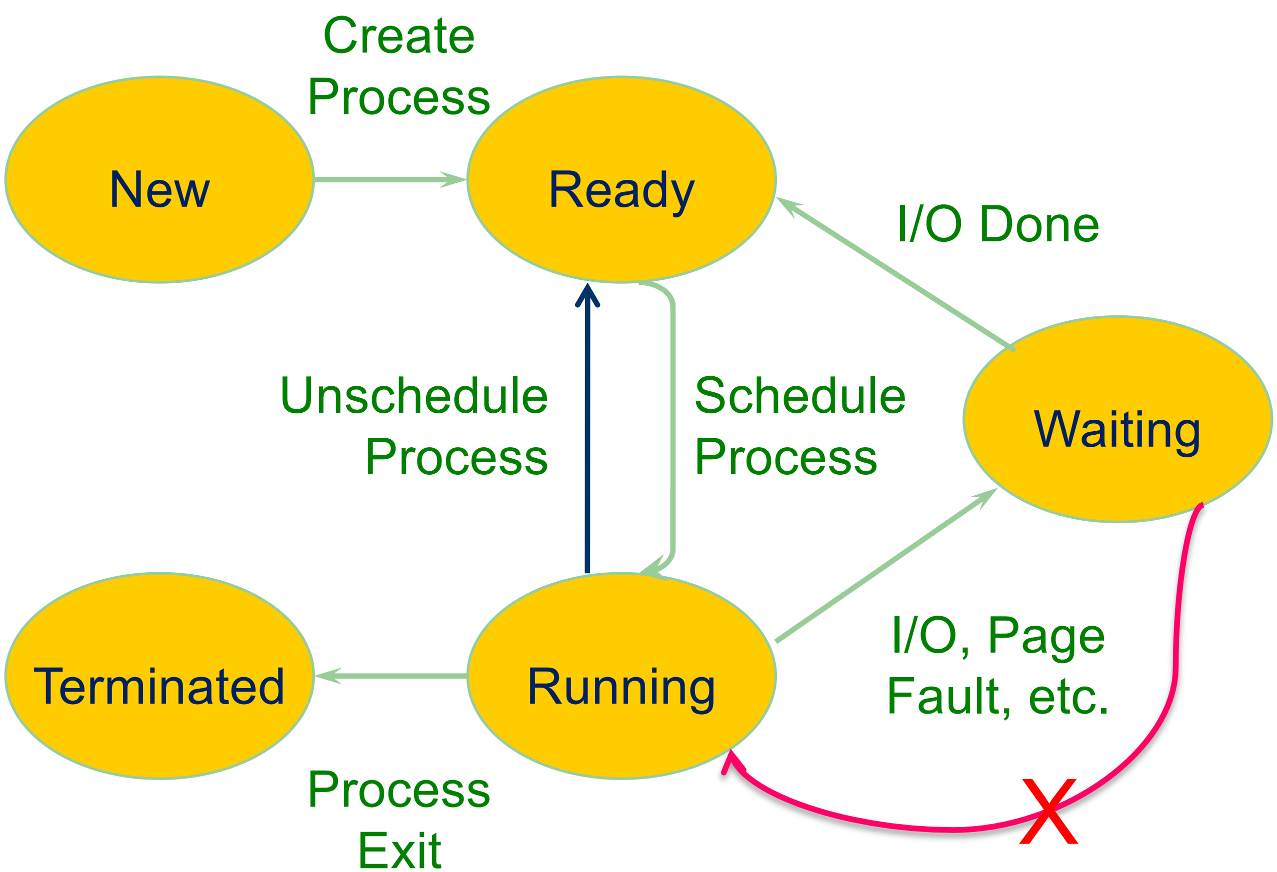
Threads: 926
Processes: 132

CPU Usage

Process State

- A process has an execution state that indicates what it is currently 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: 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)
- As a process executes, it moves from state to state
  - Linux “ps”: STAT column indicates execution state

Process State Graph



Questions

- What state do you think a process is in most of the time?
- For a uni-processor machine, how many processes can be in running state?
- How many processes can a system support?
- Benefit of multi-core (multiple CPUs)?
  - Analogy: Think of a crowded gym with only one or two treadmills

So What Is A Process?

- It's one executing instance of a "program"
- It's separate from other instances

- It can start ("launch") other processes
- It can be launched by them

So What's In A Process? And Why?

- Process State
  - new, ready, running, waiting, halted;
- Program Counter
  - the address of the next instruction to be executed for this process;
- CPU Registers
  - index registers, stack pointers, general purpose registers;
- CPU Scheduling Information
  - process priority and pointer;
- Memory Management Information
  - base/limit information, virtual→physical mapping, etc
- Accounting Information
  - time limits, process number; owner
- I/O Status Information
  - list of I/O devices allocated to the process;

Now how about this?

```
int myval;
int main(int argc, char *argv[])
{
    myval = atoi(argv[1]);
    while (1)
          printf("myval is %d, loc 0x%lx\n",
                    myval, (long) &myval);
}
```

Now simultaneously start two instances of this program

- Myval 5
- **Myval 6**
- What will the outputs be?

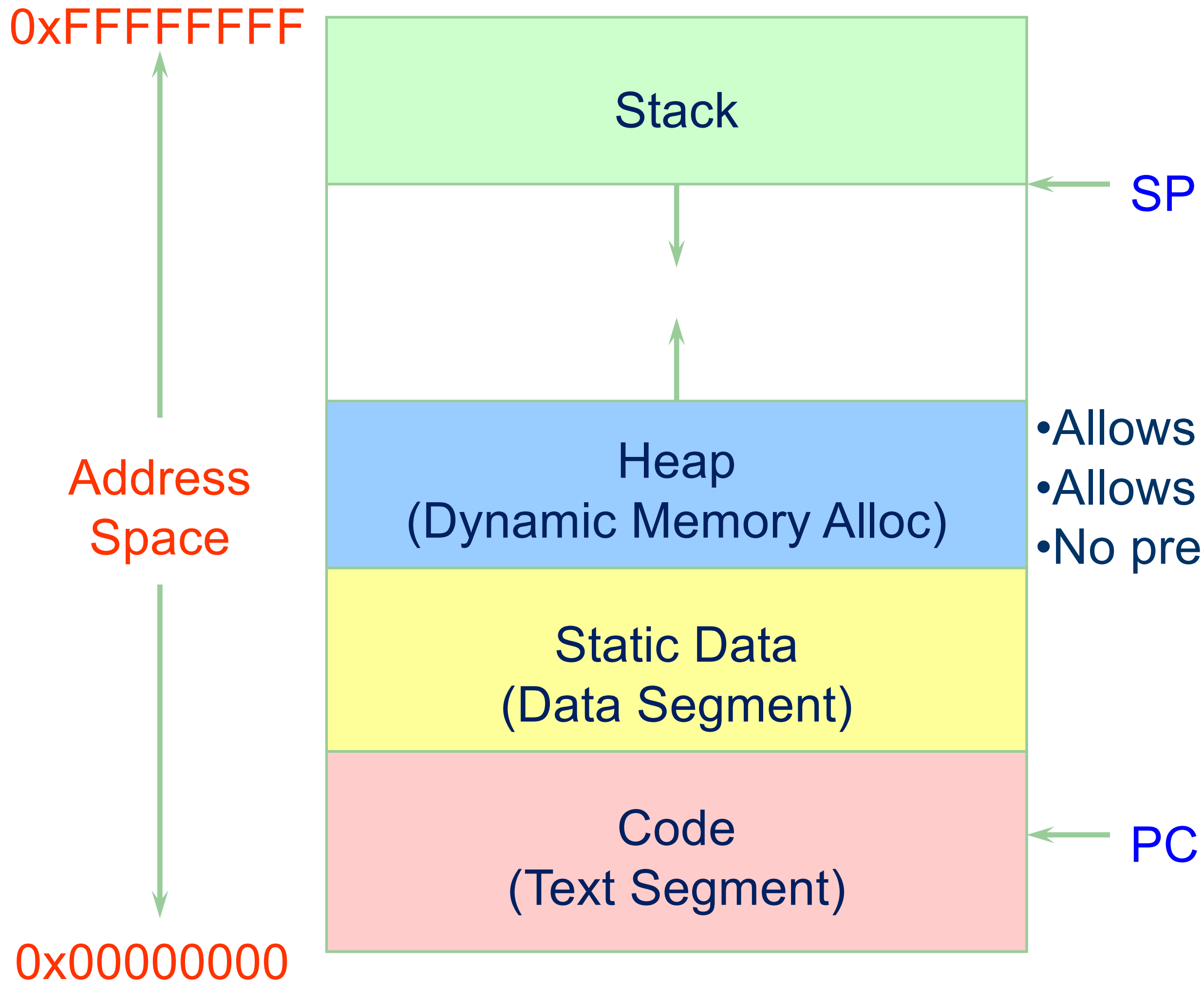
Here’s The Output

```
vivek@office-redhat-71: /home/v
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
myval 5, loc 0x804968c
```

Instances Of Programs

- The address was always the same
  - But the values were different
- Implications ?
  - the programs aren't seeing each other
  - But they think they're using the same address
- Conclusion
  - addresses are not absolute
- How?
  - memory mapping
- What's the benefit?

Process Address Space



- Allows stack growth
- Allows heap growth
- No predetermined division

Process Data Structures

How does the OS represent a process **in the kernel**?

- Process Control Block (PCB)
  - Contains all of the info about a process
  - Where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
    - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

Process Control Block (PCB)

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Fields of a process table entry

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 */

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

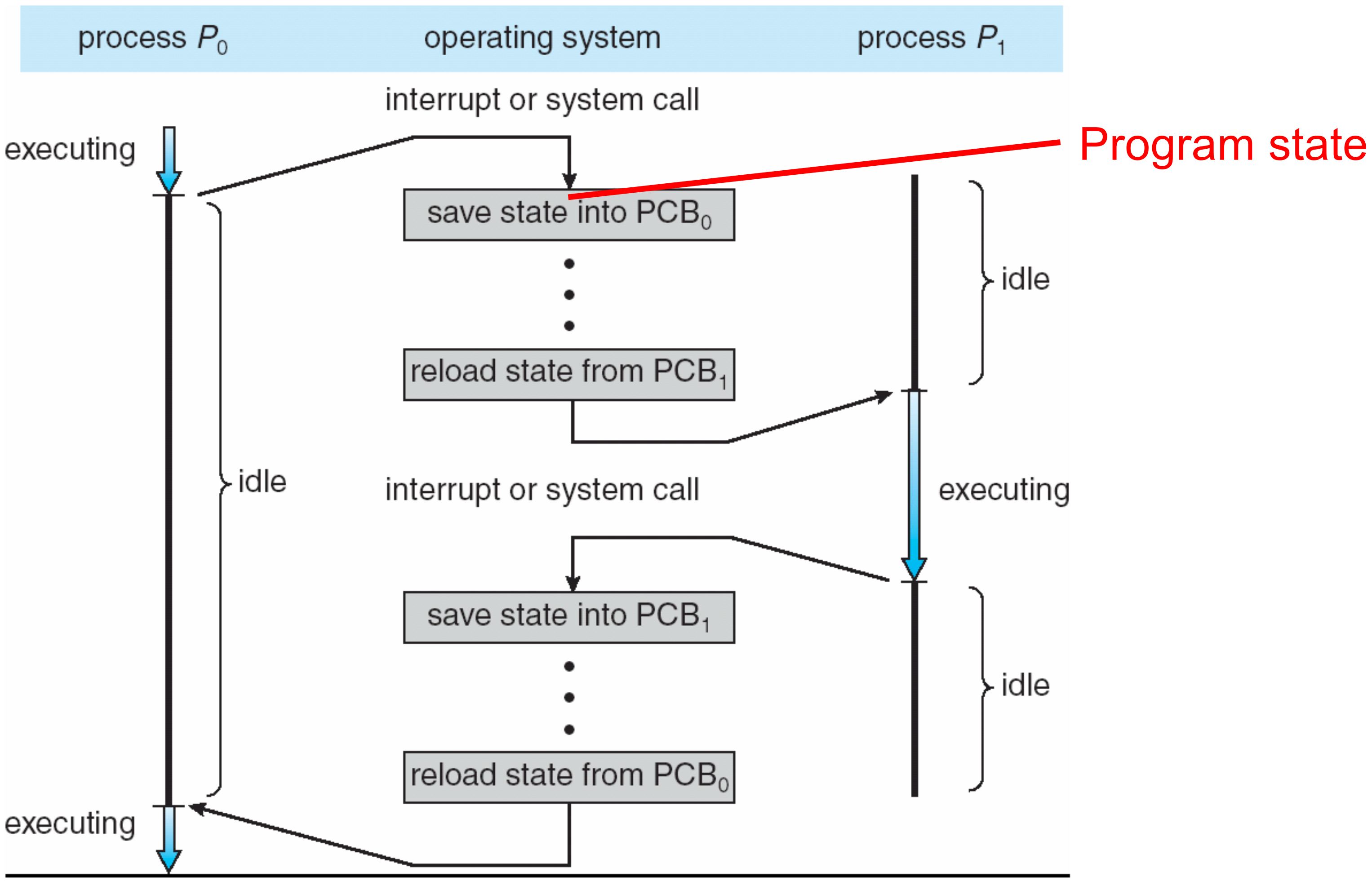
struct proc (Solaris) (3)

```
        /*
         * 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;
```

Context Switch

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
- The process of changing the CPU hardware state from one process to another is called a context switch
  - This can happen 100 or 1000 times a second!

CPU Switch From Process to Process



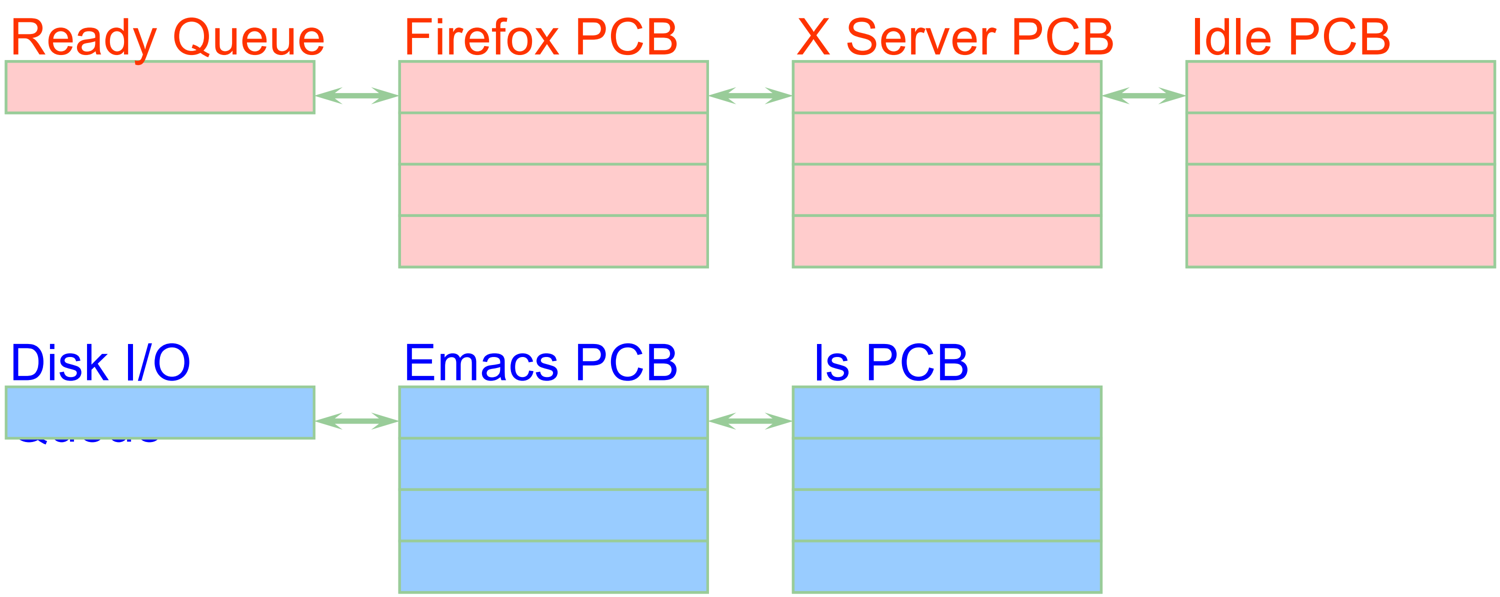
Why do you need save the program state? What program state?

Process Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the status of all processes in the system
- Typically, the OS has one queue for each status (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 unlinked from one queue and linked into another

Process Queues



Console Queue

Sleep Queue

.
.
.

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

PCBs and Queues

- PCBs are data structures dynamically allocated in OS memory (user processes cannot access it)
- When a process is created, the OS allocates a PCB for it, initializes it, and places it on the ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated

Process Creation: exec()

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

```
int exec(char *prog, char *argv[])
```

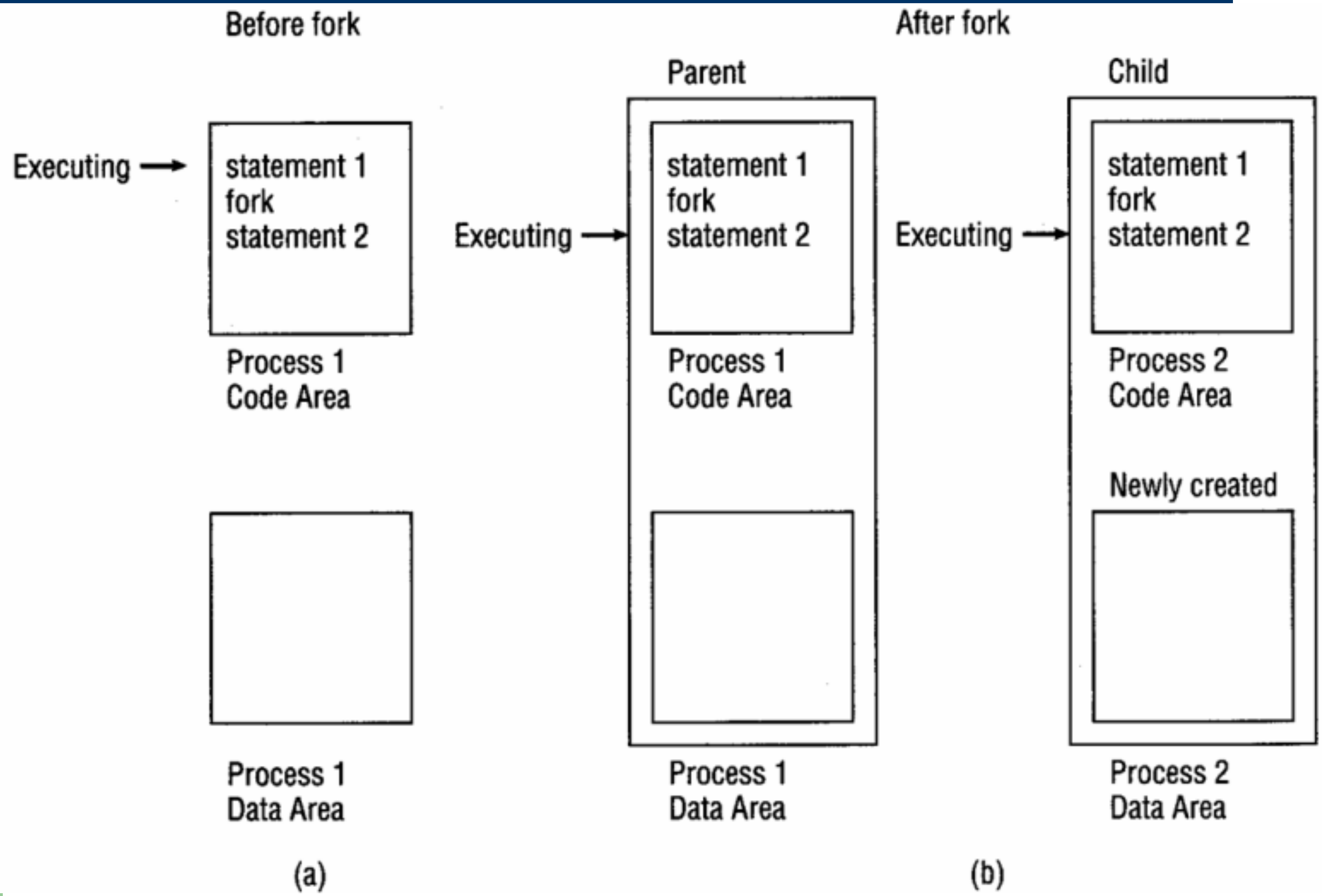
- exec()
  - Stops the current process
  - Loads the program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

Process Creation: fork()

#include <sys/types.h>
#include <unistd.h> pid_t fork(void);

- **fork** creates a child process
  - differs from the parent process only in its PID and PPID,
  - its resource utilizations are set to 0.
- RETURN VALUE
  - On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution
  - On failure, a -1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.

Fork() Semantics



An Example using Fork()

Parent alone executes this

```
pid=fork();
if (pid == 0) {
      /* child code here  */
} else {
      /* parent code here */
}
```

Child and parent both begin executing simultaneously here.

Process Creation: fork()

```
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

alpenglow (18) ~/tmp> cc t.c

alpenglow (19) ~/tmp> a.out

My child is 486

Child of a.out is 486

Duplicating Address Spaces

child_pid = 486

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

Parent

child_pid = 0

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

Child

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

Why fork()?

- 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
  }
}
```

Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), NT: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete

wait() a second…

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use wait() (WaitForSingleObject)
  - Suspends the current process until a child process ends
  - waitpid() suspends until the specified child process ends

Unix Shells

```
while (1) {
   char *cmd = read_command();
   int child_pid = fork();
   if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes,
        redirection, etc.
        exec(cmd);
        panic("exec failed");
   } else {
        waitpid(child_pid);
   }
}
```

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)