Operating Systems (A)
(Honor Track)

Lecture 6: Processes

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

This Lecture

Processes

What is a process

Processes in the kernel

Working on processes

Buzz Words

Process

Execution state

Address space

Context switch

Process control block (PCB)

Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in this course

- Today's topics are processes and process management
  - What are the units of execution?
  - How are those units of execution represented in the OS?
  - What are the possible execution states of a process?
  - How does a process move from one state to another?

This Lecture

Processes

What is a process

Processes in the kernel

Working on processes

Users, Programs

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

- Then what is a process?

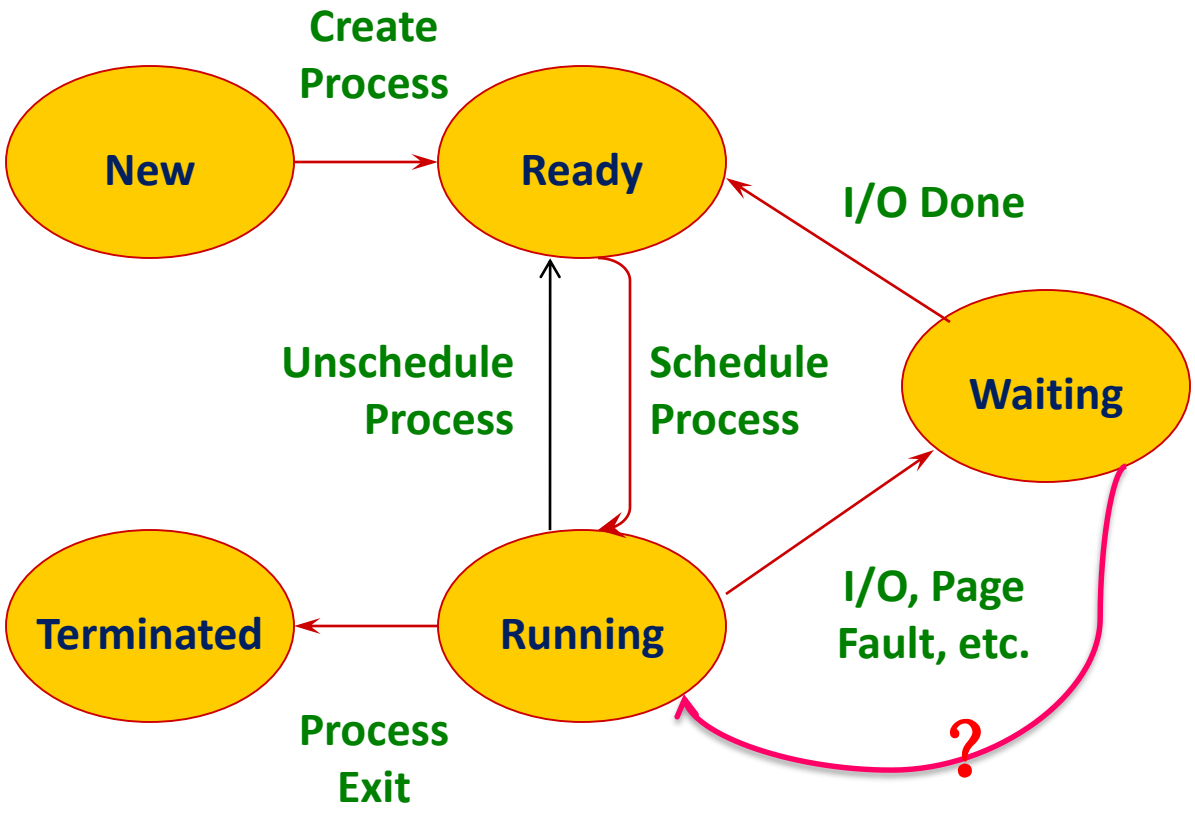
The Process

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

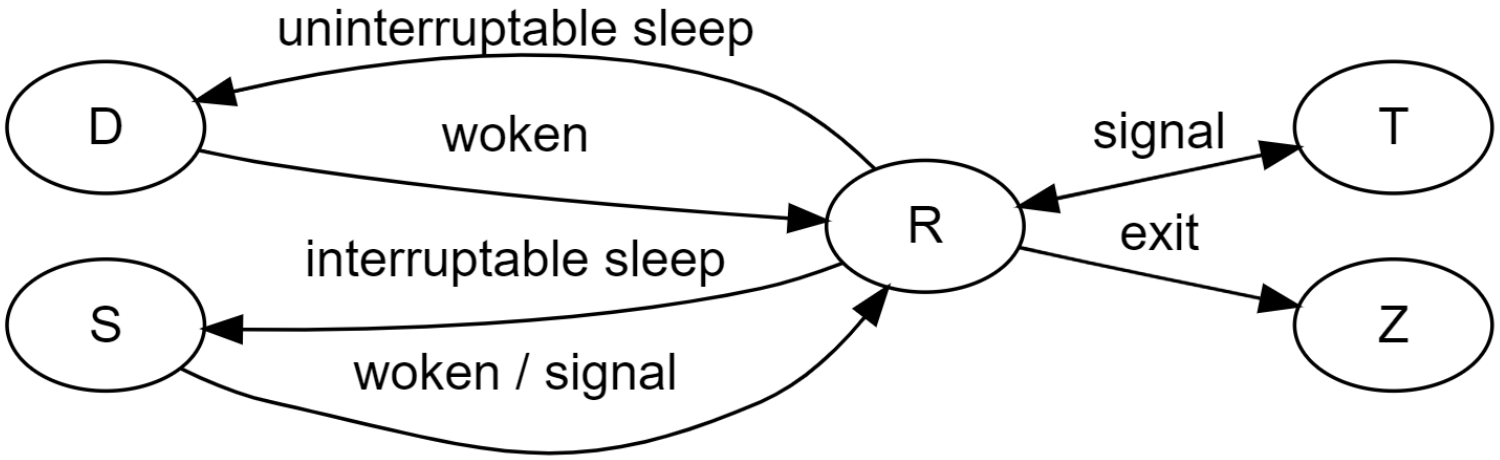
Process State

- 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

Process State Graph



Linux Processes



Linux process state codes

- R - running or runnable (on run queue)
- D - uninterruptible sleep (usually IO)
- S - interruptible sleep (waiting for an event to complete)
- Z - defunct/zombie, terminated but not reaped by its parent
- T - stopped, either by a job control signal or because it is being traced
- [...]

Windows Processes

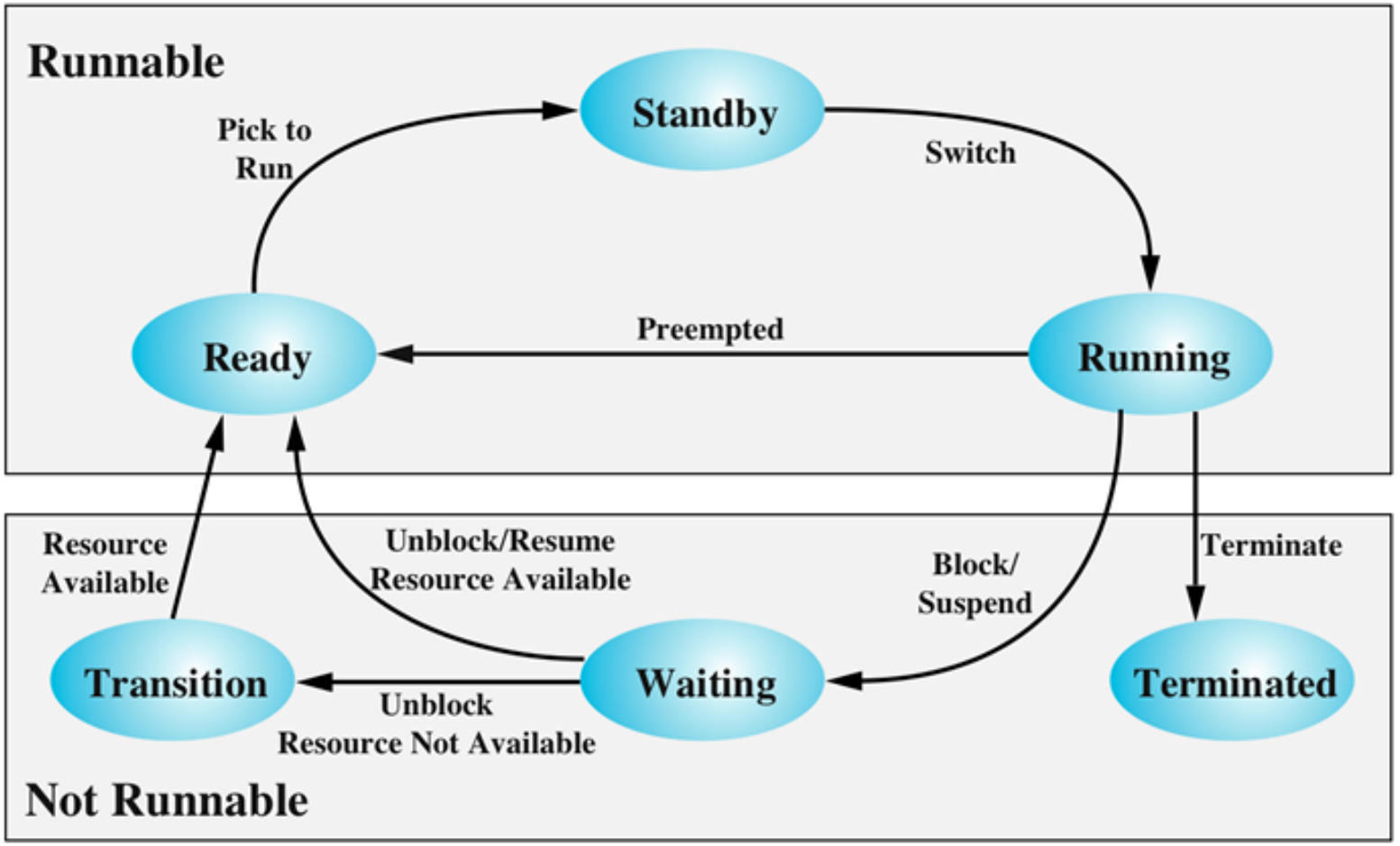


Figure 4.14 Windows Thread States

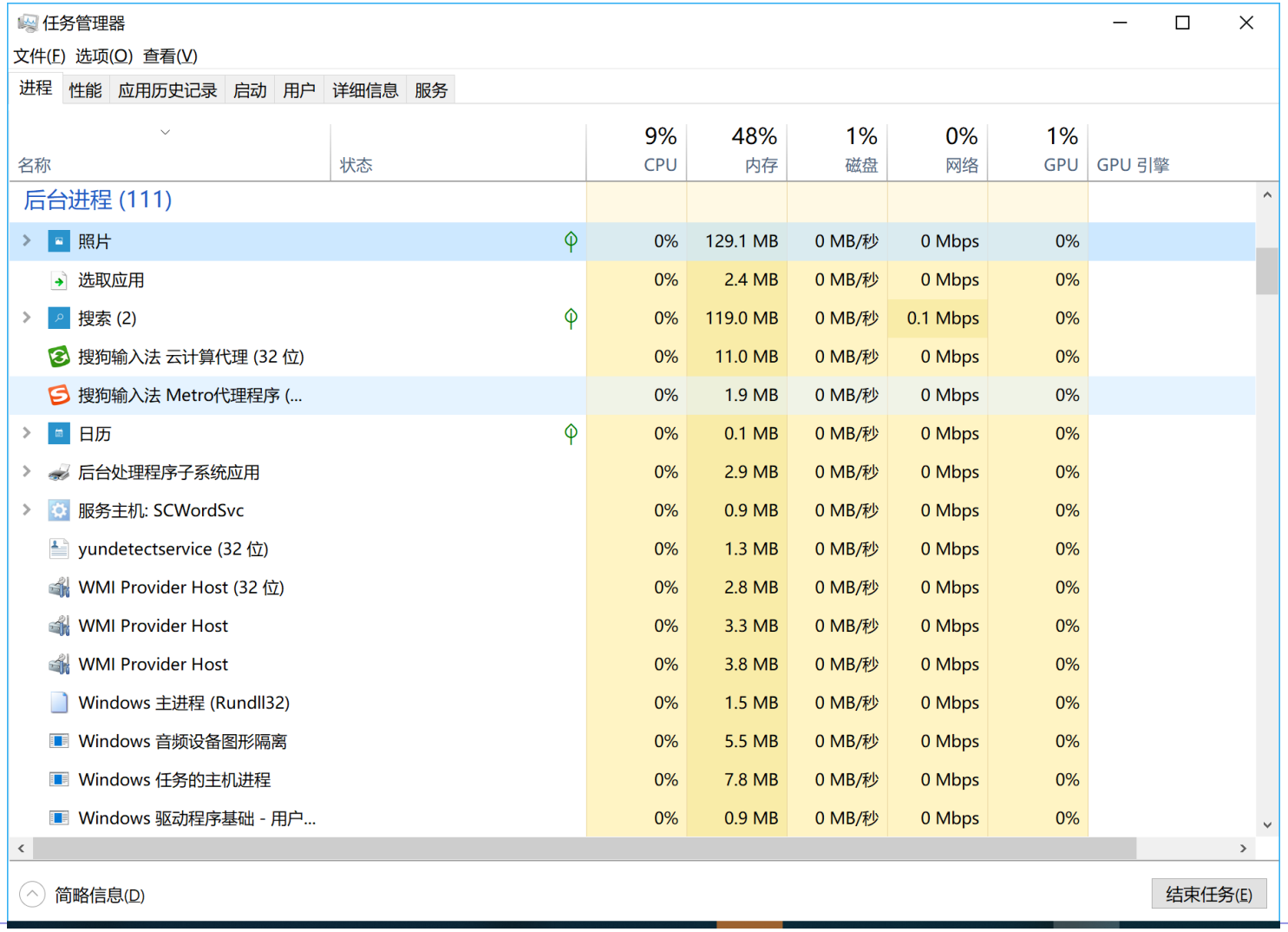
Questions

- 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?
- Benefits of multi-core?

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
- …

Windows Task Manager (Default)



Linux Example: top (Default)

wangtao@wangtao-Linuxv: ~

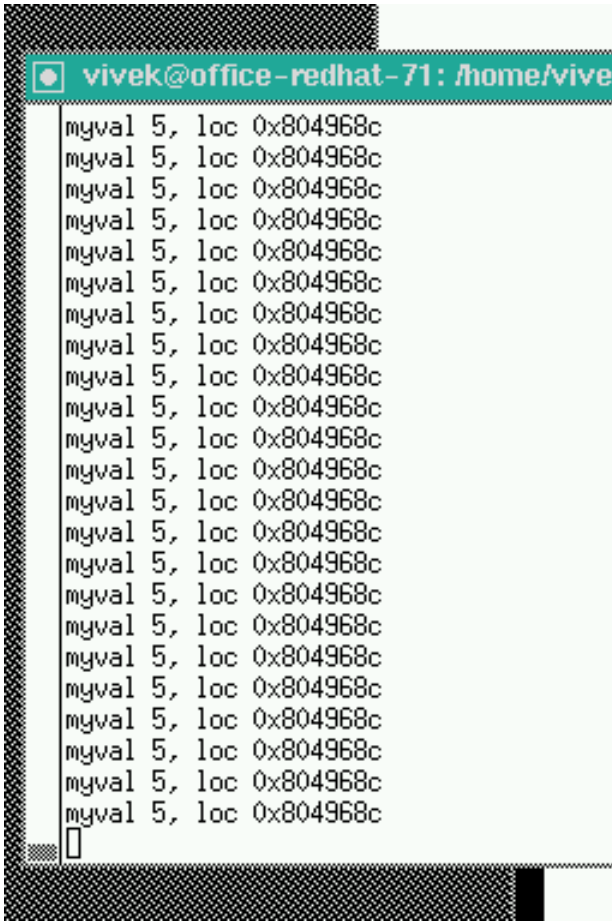
```
top - 17:02:03 up 7 min,  2 users,  load average: 0.04, 0.05, 0.05
Tasks: 155 total,   1 running, 154 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3%us,  1.0%sy,  0.0%ni, 98.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   4127876k total,   649684k used,  3478192k free,    52788k buffers
Swap:  4192252k total,        0k used,  4192252k free,   348500k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1085 root      20   0  225m  50m  12m S    2  1.2   0:03.59 Xorg
 1649 wangtao   20   0  338m  56m  33m S    1  1.4   0:01.79 unity-2d-shell
 1639 wangtao   20   0  154m  13m  10m S    0  0.3   0:00.41 metacity
 1650 wangtao   20   0  162m  26m  19m S    0  0.7   0:00.33 unity-2d-panel
 1677 wangtao   20   0  157m  24m  16m S    0  0.6   0:00.59 nautilus
 1762 wangtao   20   0 99632  16m  11m S    0  0.4   0:00.37 unity-panel-ser
    1 root      20   0  3516 1948 1312 S    0  0.0   0:00.67 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.06 ksoftirqd/0
    5 root      20   0     0    0    0 S    0  0.0   0:00.31 kworker/u:0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.01 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
   10 root      20   0     0    0    0 S    0  0.0   0:00.02 ksoftirqd/1
   11 root      20   0     0    0    0 S    0  0.0   0:00.14 kworker/0:1
   12 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
   13 root       0 -20     0    0    0 S    0  0.0   0:00.00 cpuset
```

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 in two terminals
  - Myval 5
  - **Myval 6**
- What will the outputs be?

Here's The Output

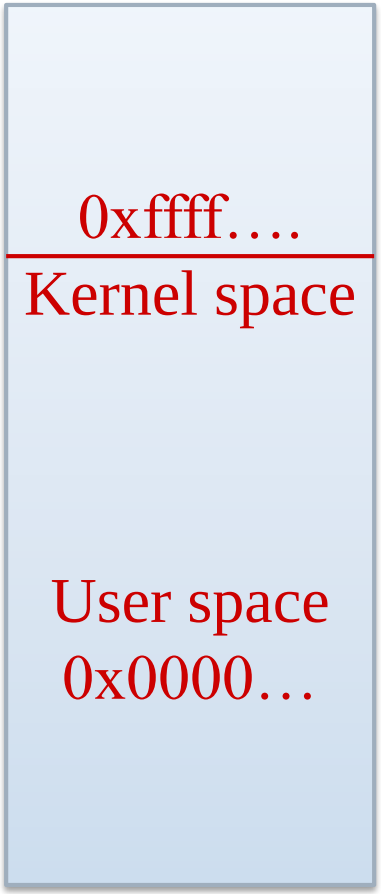


Instances of Programs

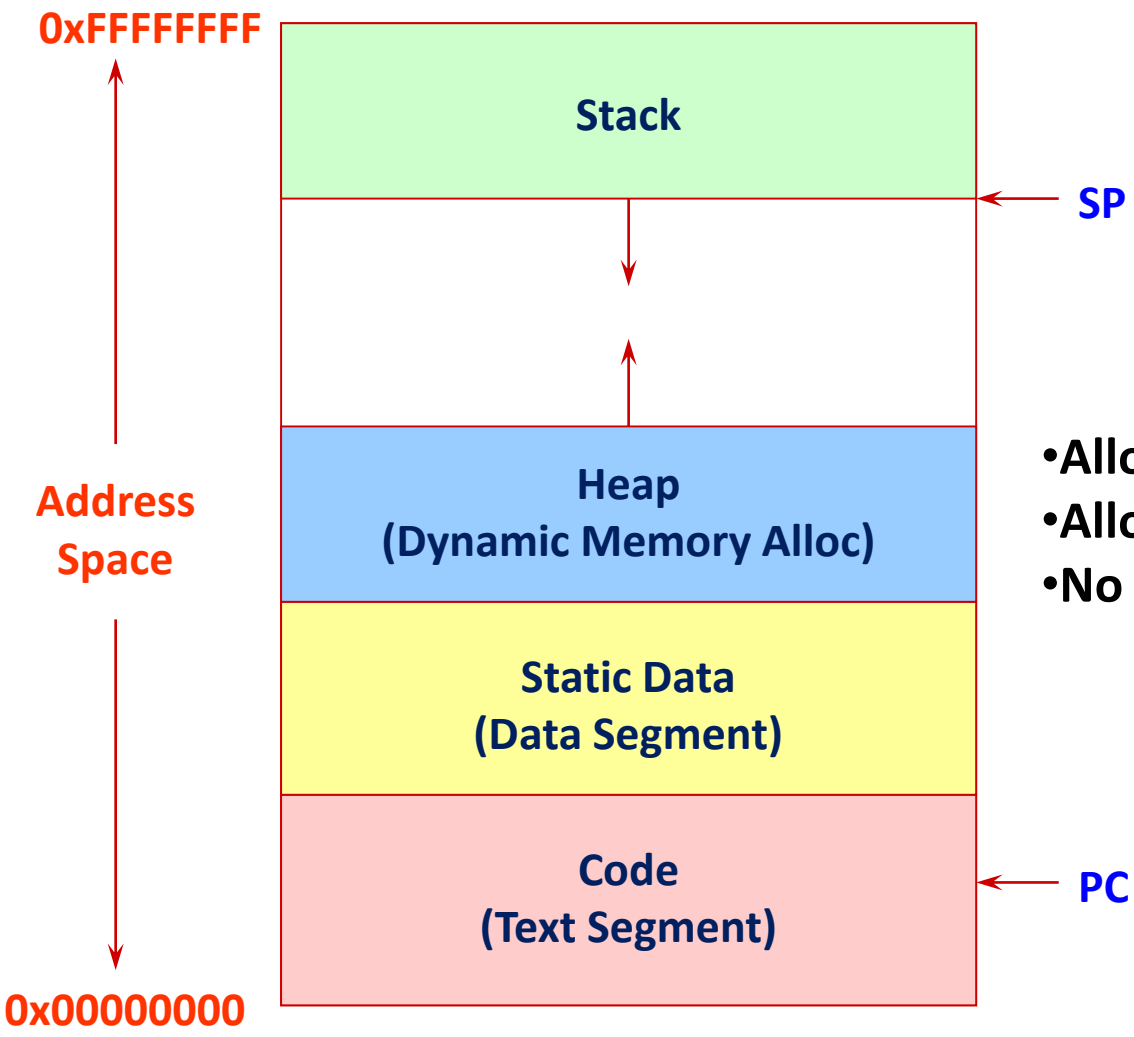
- The address was always the same
  - But the values were different
- Implications ?
  - The programs aren't seeing each other
  - But they think they're using the same address
- Conclusion
  - Addresses are not absolute
- How?
  - Memory mapping
- What's the benefit?

Address Space

- Each process has its own address space
- One (common) approach
  - Kernel is in the high memory
  - User is in the low memory
- What restrictions apply?



Process Address Space (Ignoring Kernel)



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

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

This Lecture

Processes

What is a process

Processes in the kernel

Working on processes

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	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
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		

Figure 2-4. Some of the fields of a typical 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;
      struct  proc    *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)

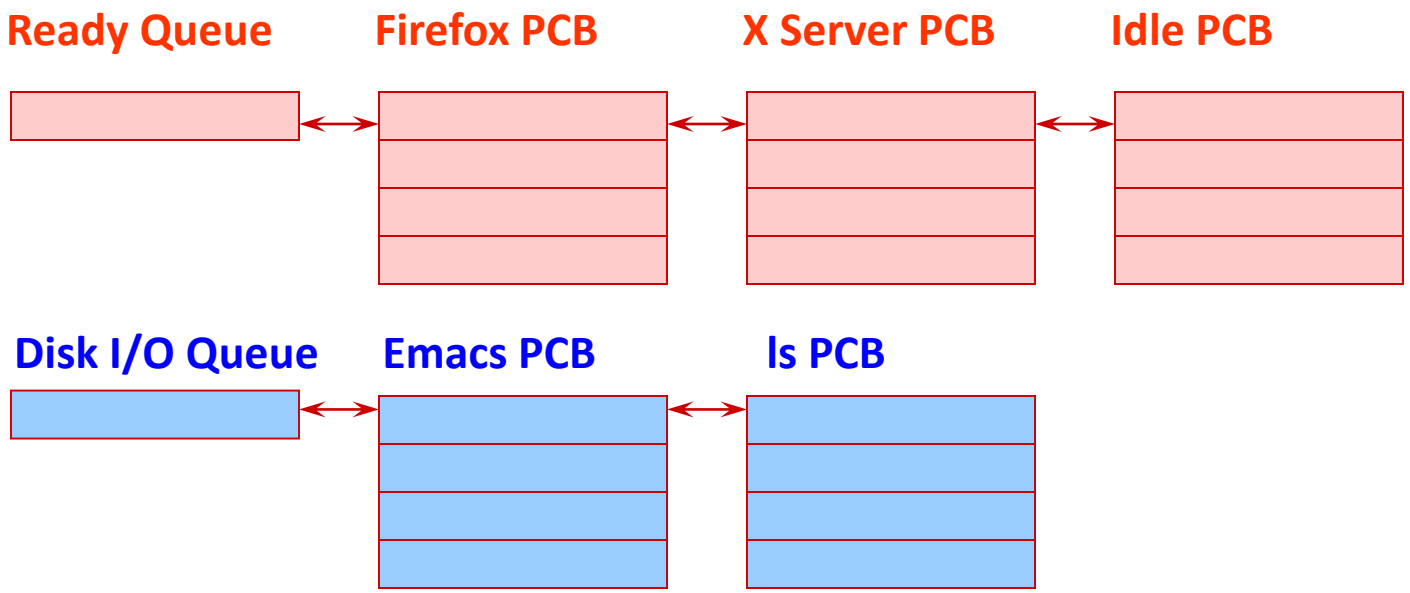
```
    /*
     * C2 Security  (C2_AUDIT)
     */
    caddr_t p_audit_data;         /* per process audit structure */
    kthread_t      *p_aslwptp;    /* thread ptr representing "aslwp" */
#if defined(i386) || defined(__i386) || defined(__ia64)
    /*
     * LDT support.
     */
    kmutex_t p_ldtlock;           /* 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;
```

State Queues

How does the OS keep track of processes?

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

State Queues



Console Queue

Sleep Queue

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

PCBs and State Queues

- PCBs are data structures dynamically allocated in OS memory
- 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

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's PCB
  - And to which queue does the OS put the PCB?
- When the OS is ready to start executing a process, it loads the hardware registers from the values stored in that process' PCB
  - Which queue does the PCB come from?
- 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!

This Lecture

Processes

What is a process

Processes in the kernel

Working on processes

Process Creation: exec() ?

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

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

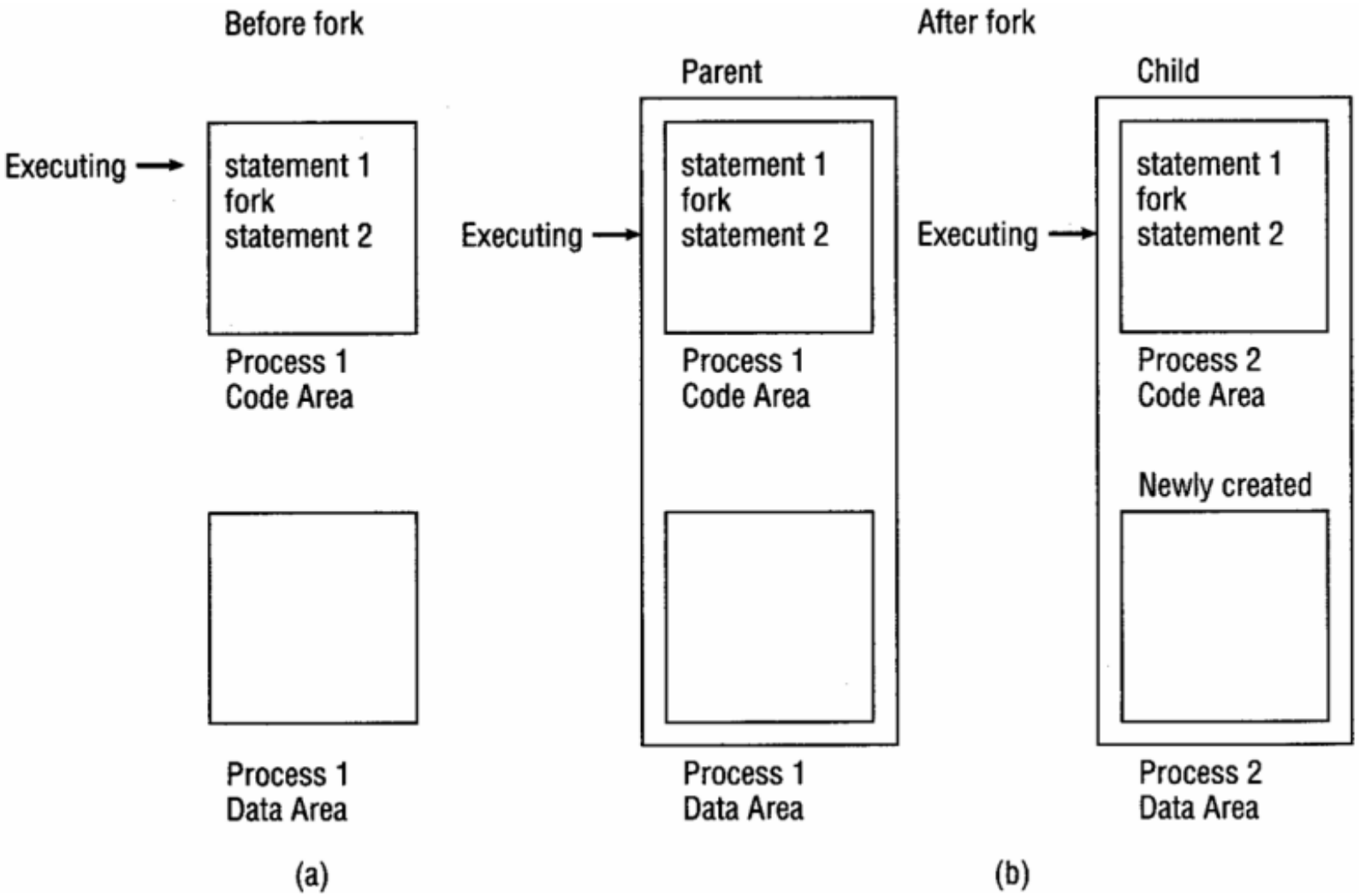
- exec()
  - Stops the current process
  - Loads the program "prog" into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

Process Creation: fork()

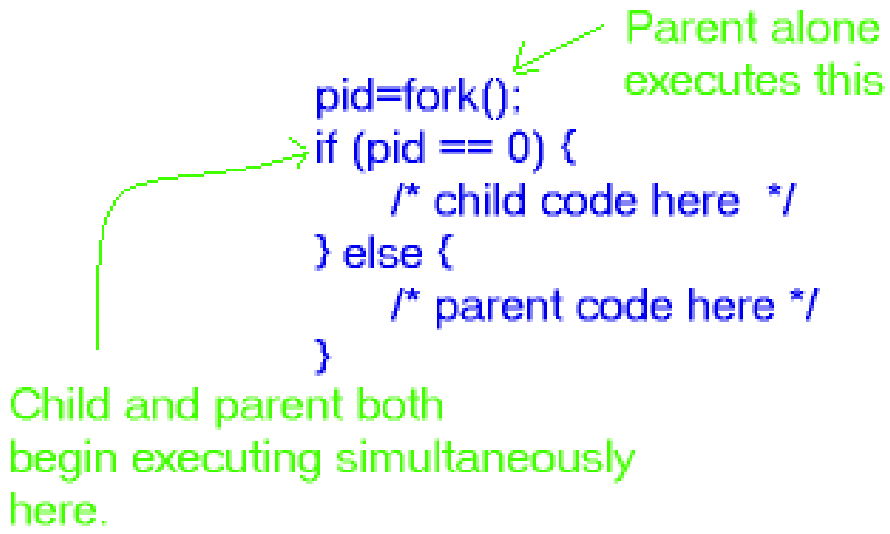
- `#include <sys/types.h>`
- `#include <unistd.h> pid_t fork(void);`

- fork creates a child process
  - differs from the parent process only in its PID and PPID,
  - its resource utilizations are set to 0.

Fork() Semantics



Using fork()



- **Return value of fork()**
  - On success, the PID of the child process is returned in the parent's thread of execution, and 0 is returned in the child's thread of execution
  - On failure, -1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.

An Example of Using fork()

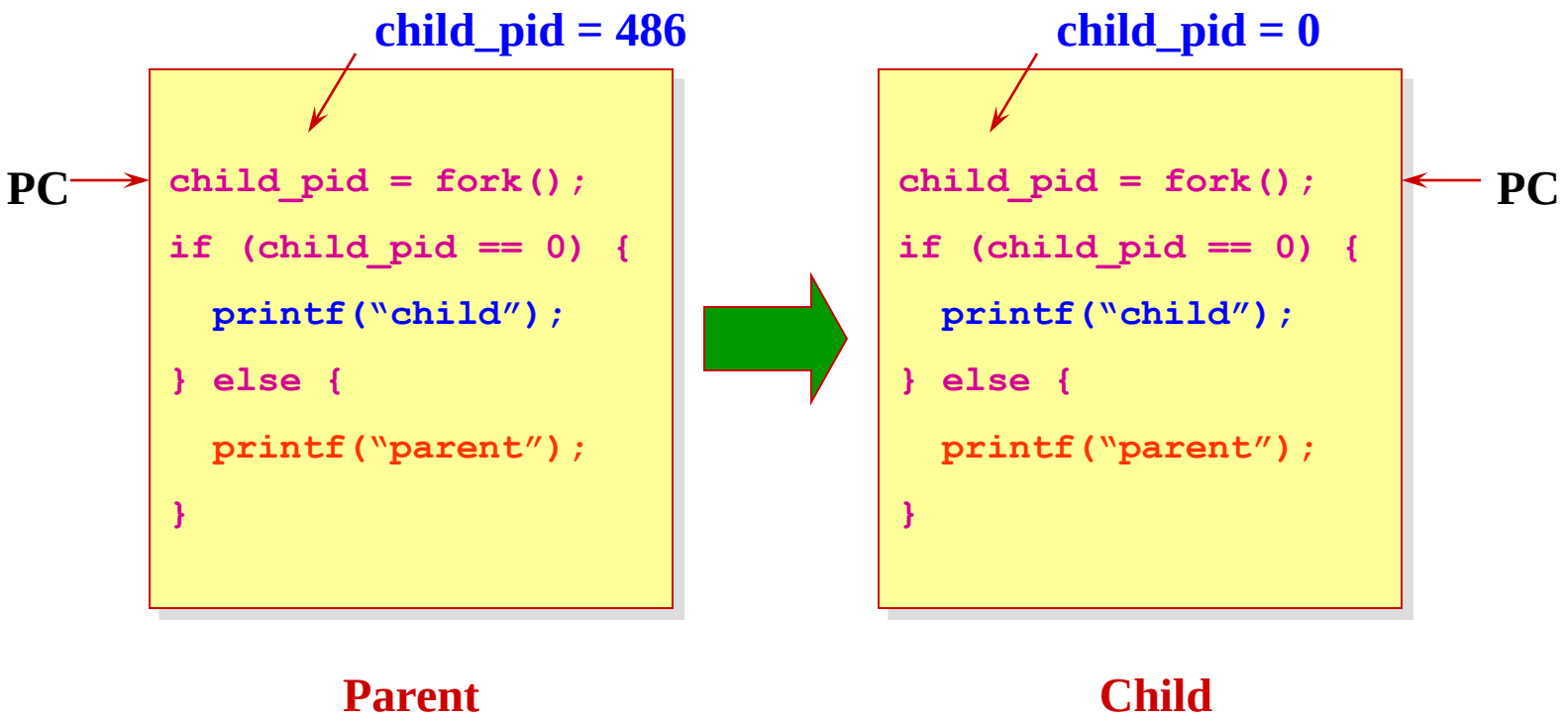
```
int main(int argc, char *argv[])
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

What does this program print?

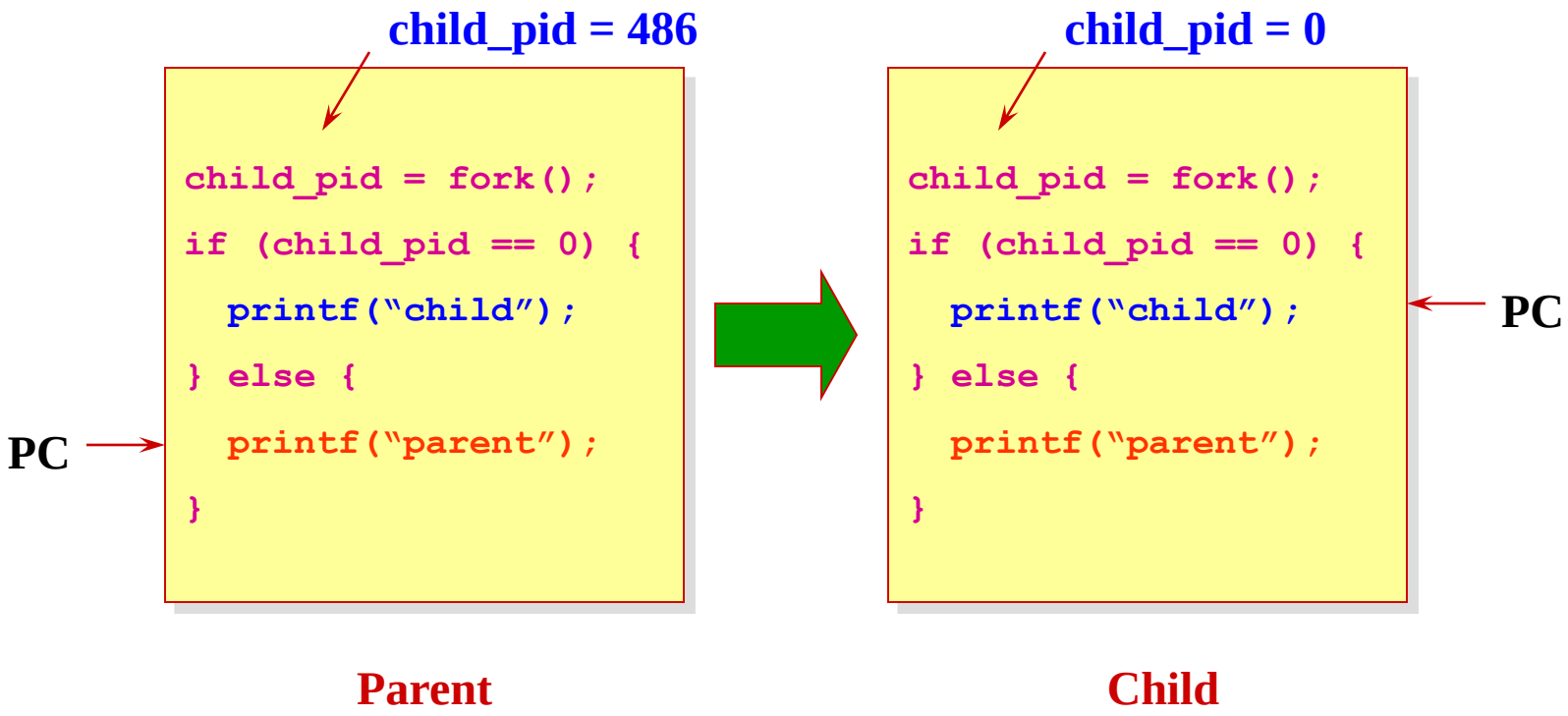
Example Output


```
> cc t.c
> ./a.out
My child is 486
Child of a.out is 486
```

Duplicating Address Spaces



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

```
> cc t.c
> ./a.out
My child is 486
Child of a.out is 486

> ./a.out
Child of a.out is 498
My child is 498
```

Why is the output in a different order?

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
    }
}
```

- Question: how to "create an instance of another program"?

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
  - Free allocated memory (and VM pages out on disk)
  - 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
- Use wait() (WaitForSingleObject)
  - Suspends the current process until a child process ends
  - waitpid() suspends until the specified child process ends

- Unix: Every process must be "reaped" by a parent
  - What do you think a "zombie" process is?
    - A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to do something to clean up its zombie children
  - What happens if a parent process exits before a child?

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);
    }
}
```

Practice

- Create *N* processes in BOTH Linux and Windows, printing “Hello world! The process ID is xxx” (xxx should be the newly created process’ ID)
  - N = 1, 10, 100, 1000, 10000, 100000, 1000000
- Create a zombie process

Summary

- Processes
  - What is a process
  - Processes in the kernel
  - Working on processes
- Next lecture: Threads