

# Operating Systems

## CSCI 3150

### *Lecture 3: Processes*

Hong Xu

# Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in this course
  - You can rest assured that they will be covered in the exams
- 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?

# 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
- Then what is a process?









# The Process

- The process is the OS's **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 concrete **instantiation** of the program
- A process is sometimes called a **job** or a **task** or a **sequential process**
- Real life analogy?

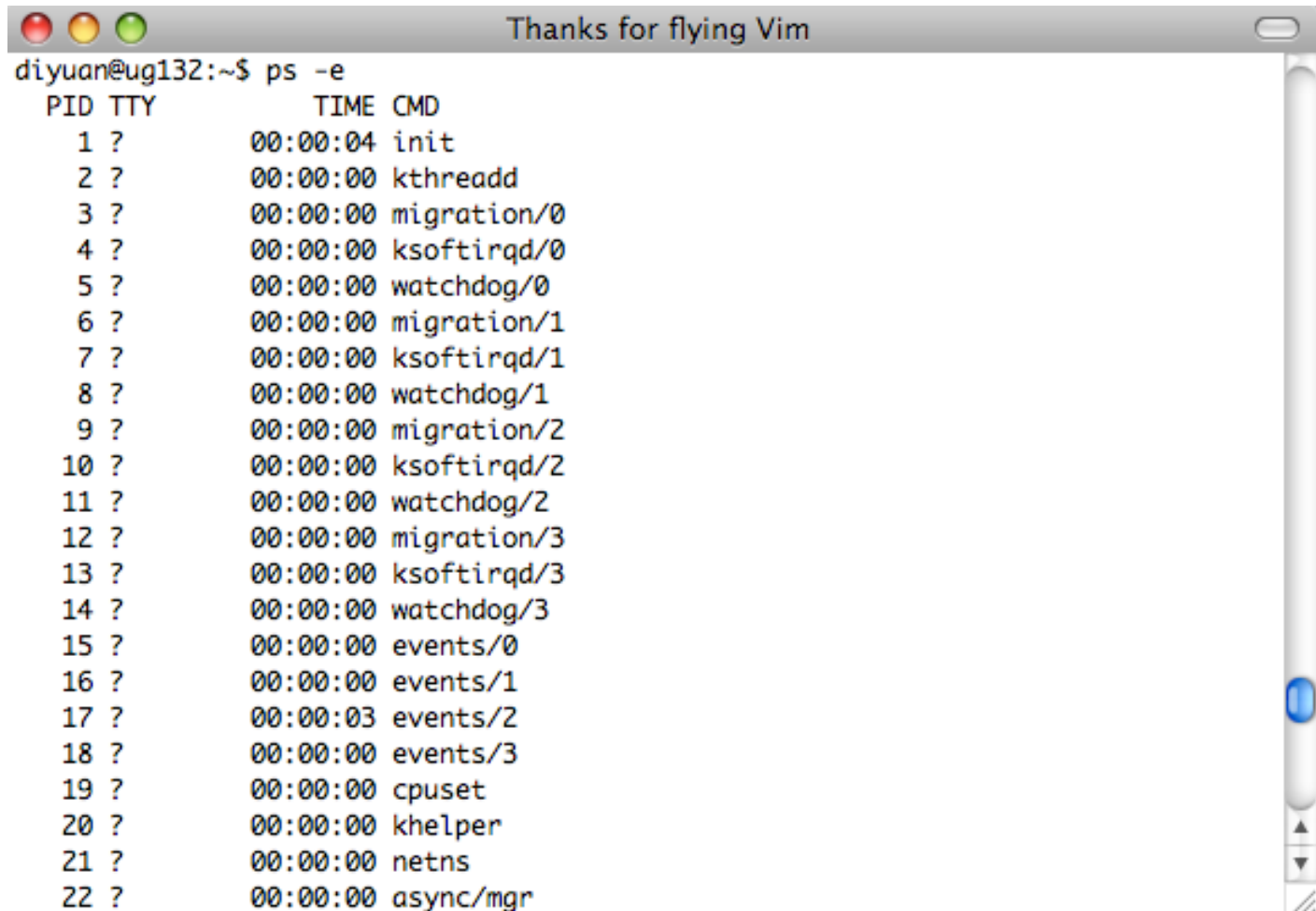
# Analogy: A robot taking CSCI3150

- **Program**: steps for attending the lecture
  - Step 1: take a school bus
  - Step 2: enter the classroom
  - Step 3: listen (or sleep)
- **Process**: attending this particular lecture right now
  - Action
  - You are all in the middle of a process

# MacOS example: Activity monitor

Process Name	% CPU ▾	CPU Time	Threads	Idle Wake-Ups	% GPU	GPU Time	PID	User
Microsoft Edge Helper (Rend...	2.8	1:20.50	29	58	0.0	0.00	4283	henry
 Activity Monitor	1.7	2:47.08	5	3	0.0	0.00	1706	henry
photolibraryd	0.9	56.12	8	0	0.0	0.00	525	henry
accountsd	0.9	12.62	4	0	0.0	0.00	477	henry
 Screenshot	0.8	0.26	5	1	0.0	0.00	5124	henry
Microsoft Edge Helper (GPU)	0.7	2:22.43	13	66	0.1	27.71	1907	henry
Microsoft Edge Helper (Rend...	0.7	1.27	16	0	0.0	0.00	1968	henry
 Dropbox	0.6	3:30.31	146	26	0.0	0.00	1939	henry
 WeChat	0.6	1:34.12	38	23	0.0	0.16	3925	henry
 Microsoft Edge	0.5	2:23.11	46	6	0.0	0.00	1697	henry
 iStat Menus Status	0.5	3:39.07	4	3	0.0	0.00	1930	henry
Microsoft Edge Helper	0.5	21.75	11	16	0.0	0.00	2062	henry
tccd	0.4	4.11	4	0	0.0	0.00	471	henry
mdworker_shared	0.4	0.06	3	0	0.0	0.00	5121	henry
routined	0.4	10.13	5	0	0.0	0.00	489	henry
 System Preferences	0.4	1.51	3	0	0.0	0.00	1696	henry
 Microsoft PowerPoint	0.3	29.15	45	4	0.0	0.24	4932	henry

# Linux example: ps



```
diyuan@ug132:~$ ps -e
```

PID	TTY	TIME	CMD
1	?	00:00:04	init
2	?	00:00:00	kthreadd
3	?	00:00:00	migration/0
4	?	00:00:00	ksoftirqd/0
5	?	00:00:00	watchdog/0
6	?	00:00:00	migration/1
7	?	00:00:00	ksoftirqd/1
8	?	00:00:00	watchdog/1
9	?	00:00:00	migration/2
10	?	00:00:00	ksoftirqd/2
11	?	00:00:00	watchdog/2
12	?	00:00:00	migration/3
13	?	00:00:00	ksoftirqd/3
14	?	00:00:00	watchdog/3
15	?	00:00:00	events/0
16	?	00:00:00	events/1
17	?	00:00:03	events/2
18	?	00:00:00	events/3
19	?	00:00:00	cpuset
20	?	00:00:00	khelper
21	?	00:00:00	netns
22	?	00:00:00	async/mgr

# So what is a process?

- A process is a **program in execution**
- It is one executing instance of a program
- It is separated from other instances
- It can start (“launch”) other processes
- It can be launched by them



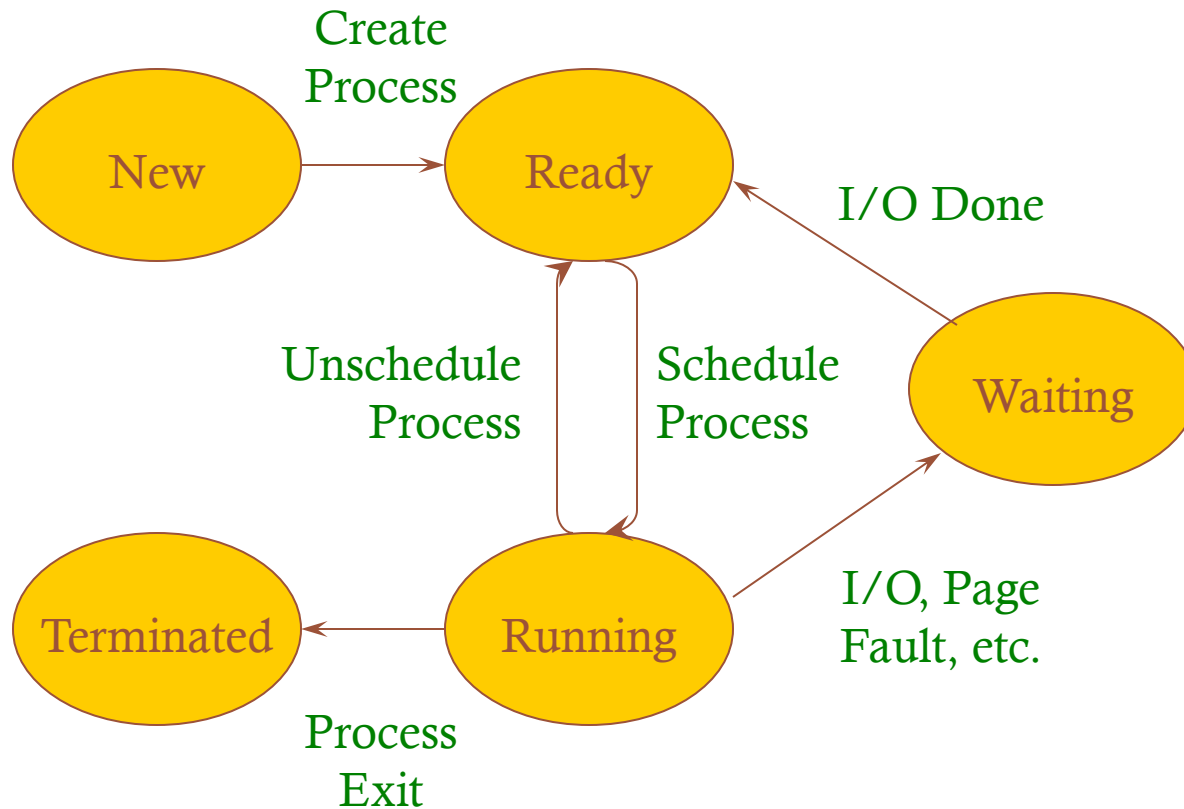
# Process State

- 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
  - Unix “ps”: **STAT** column indicates execution state

# 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?
- Benefit of multi-core?

# Process State Graph



# Process Components

- Process State
  - new, ready, running, waiting, terminated;
- 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;

# Process Components (cont.)

- 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;
- An Address Space
  - memory space visible to one 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?

D

## Output differs on your machine!



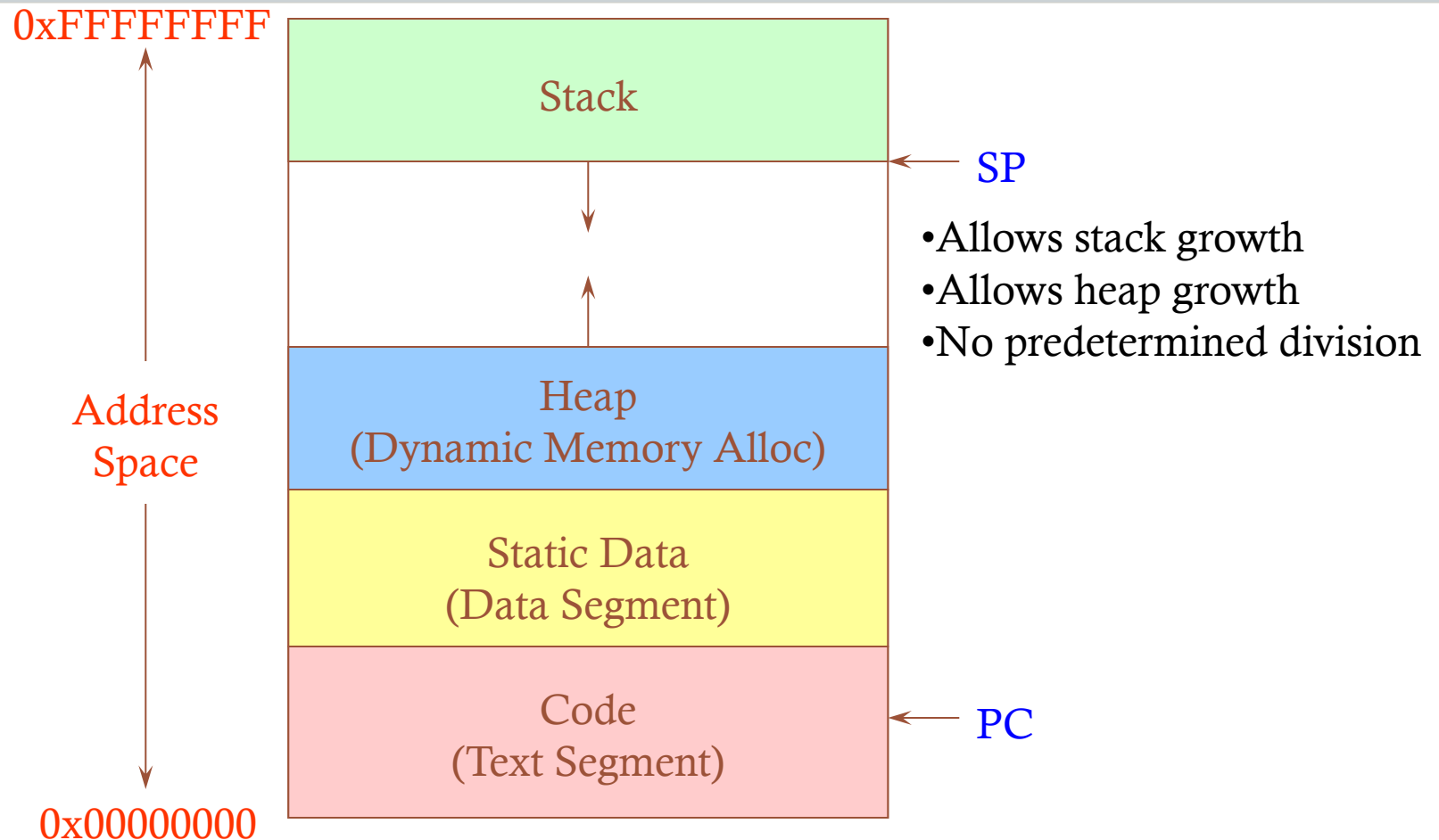
15

# 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
- Conclusions
  - addresses are not the “physical memory”
- How?
  - Memory mapping
- What is the benefit?



# Process Address Space



# Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the **Process Control Block** (PCB)
- The PCB contains all of the info about a process
- The PCB also is 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 state it was in when the process was switched out of the hardware

# PCB Data Structure

- The PCB contains a huge amount of information in one large structure
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Pointers for state queues
  - Etc.

# 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_crlck; /* 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_pgdp; /* 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 be held. */
    ushort_t p_pad1; /* unused */
    uint_t p_flag; /* protected while set. */
```

```
    /* flags defined below */
```

```
    clock_t p_utm; /* user time, this process */
    clock_t p_stm; /* 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_sighdr; /* 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_ftset_t p_ftmask; /* 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_srwhchan_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_aslwp; /* 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_notifsig; /* 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 initialization of robust locks.
 */
kmutex_t p_lcp_mutexinitlock;
utrap_handler_t *p_utrap; /* 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!

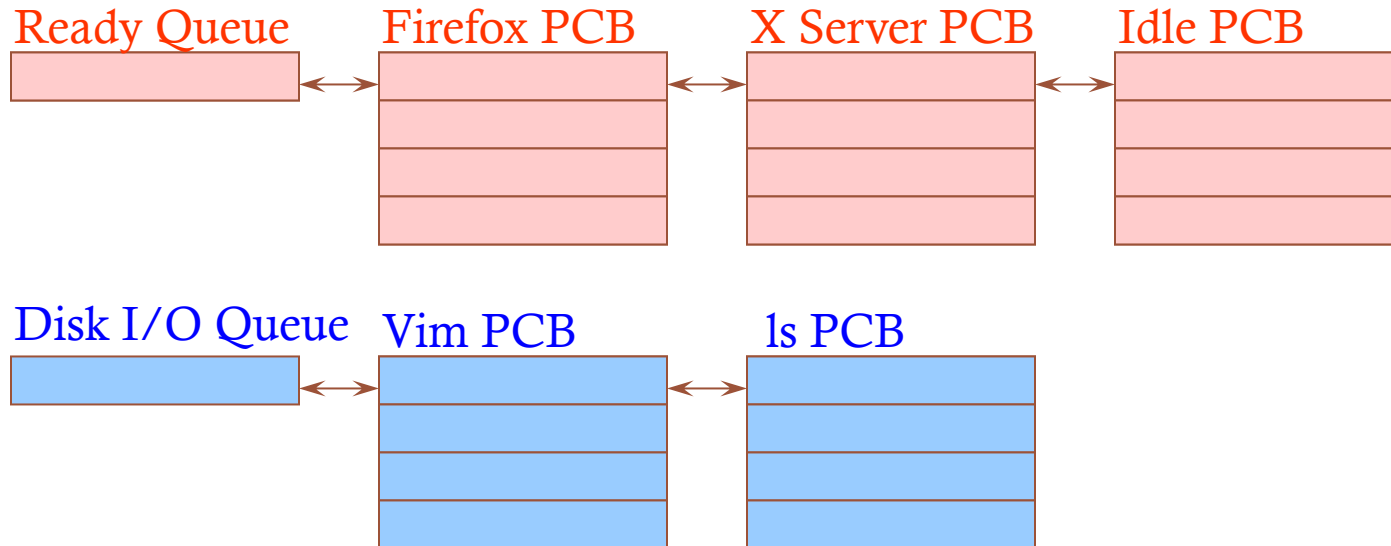
# State Queues

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

# Process Creation

- A process is created by another process
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first (userspace) process? (Unix: init (PID 1))
- In some systems, the parent defines (or donates) 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 (or both)

# Process Creation: Windows

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

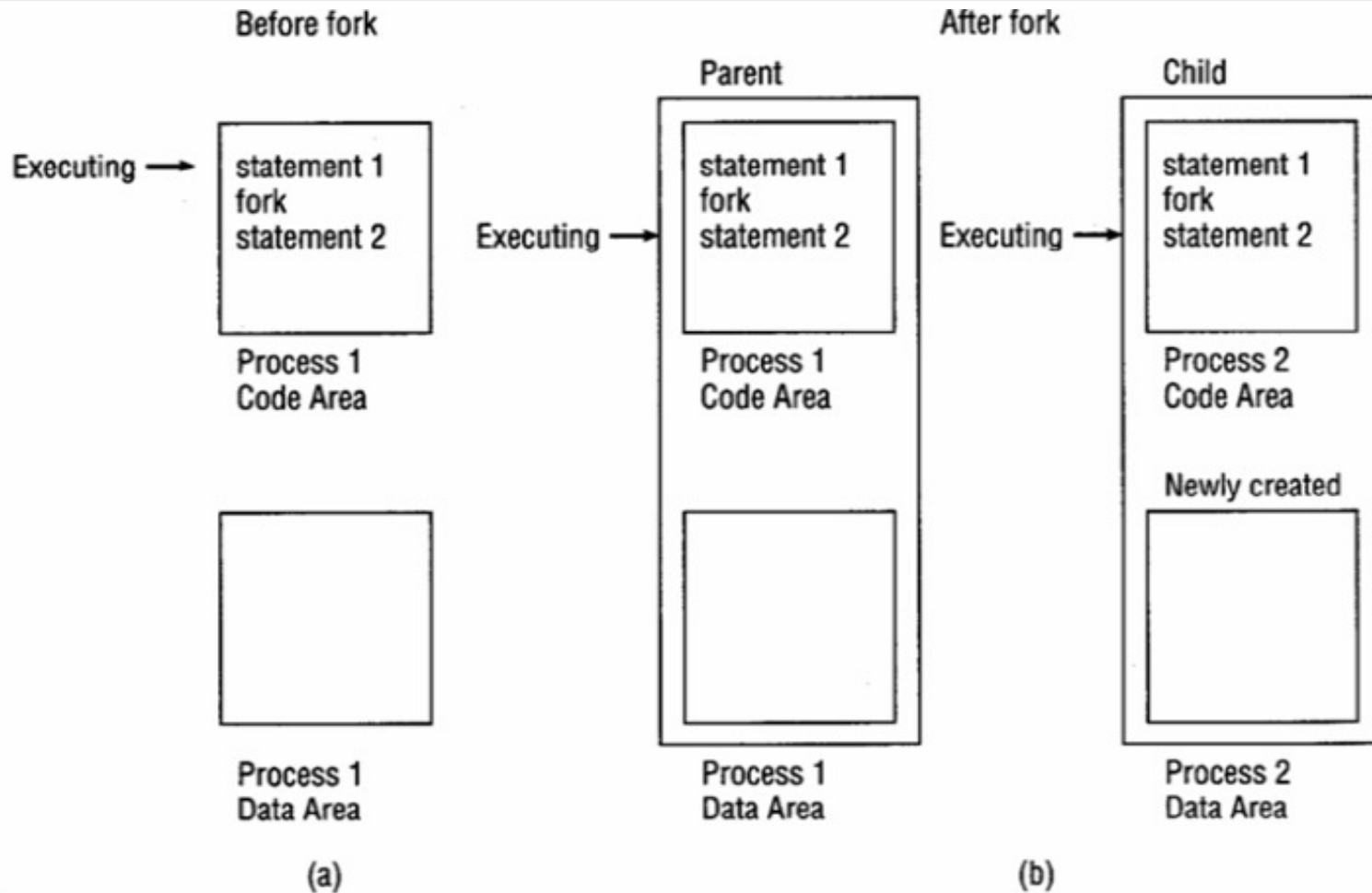
**BOOL CreateProcess(char \*prog, . . . .)** (simplified)

- `CreateProcess`
  - Creates and initializes a new **PCB**
  - Creates and initializes a new address space
  - Loads the program specified by “prog” into the address space
  - Initializes the hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the **ready** queue

# Process Creation: Unix

- In Unix, processes are created using `fork()`  
`int fork(void)`
- **`fork(void)`**
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns **twice**
  - Returns the child's PID to the parent, "0" to the child
  - Huh?

# fork() semantics



# 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

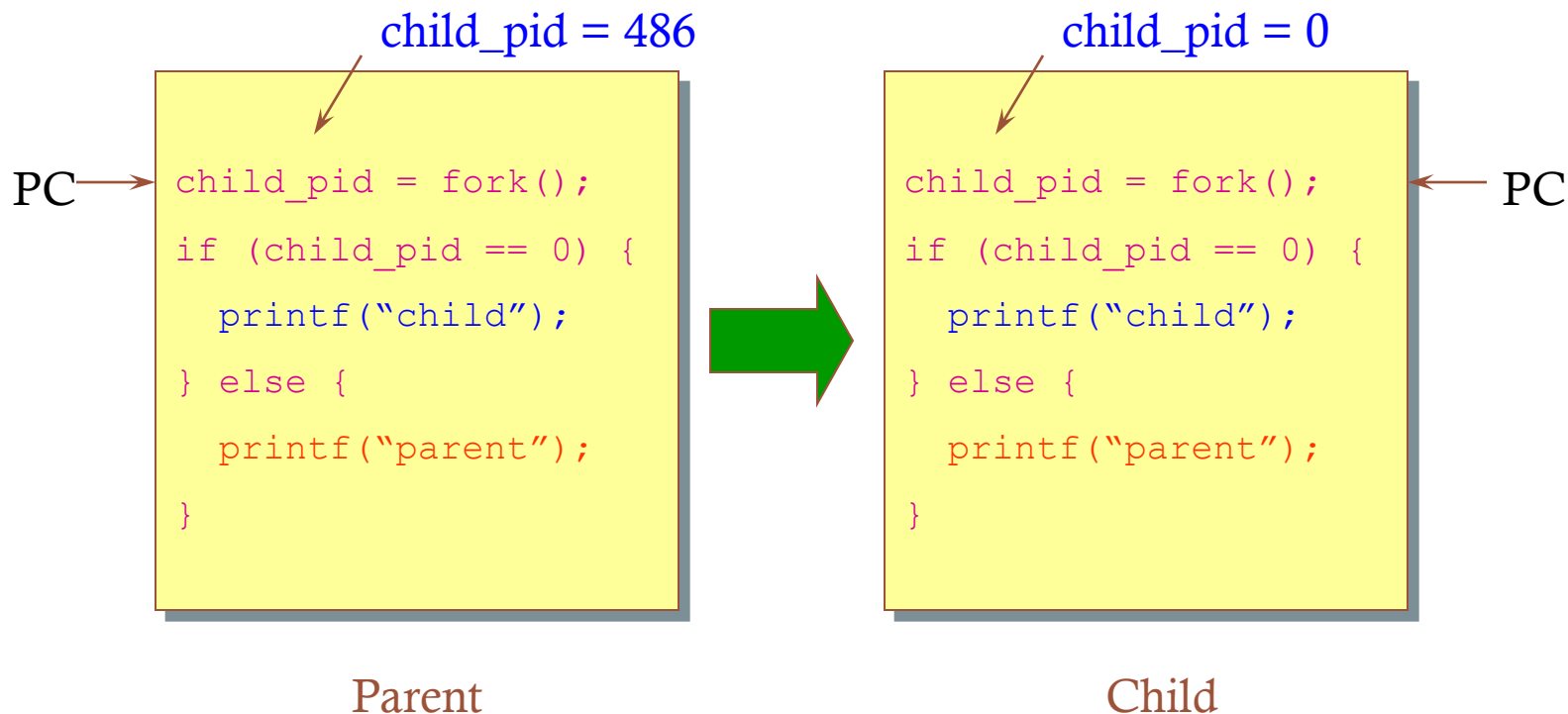
---

My child is 486

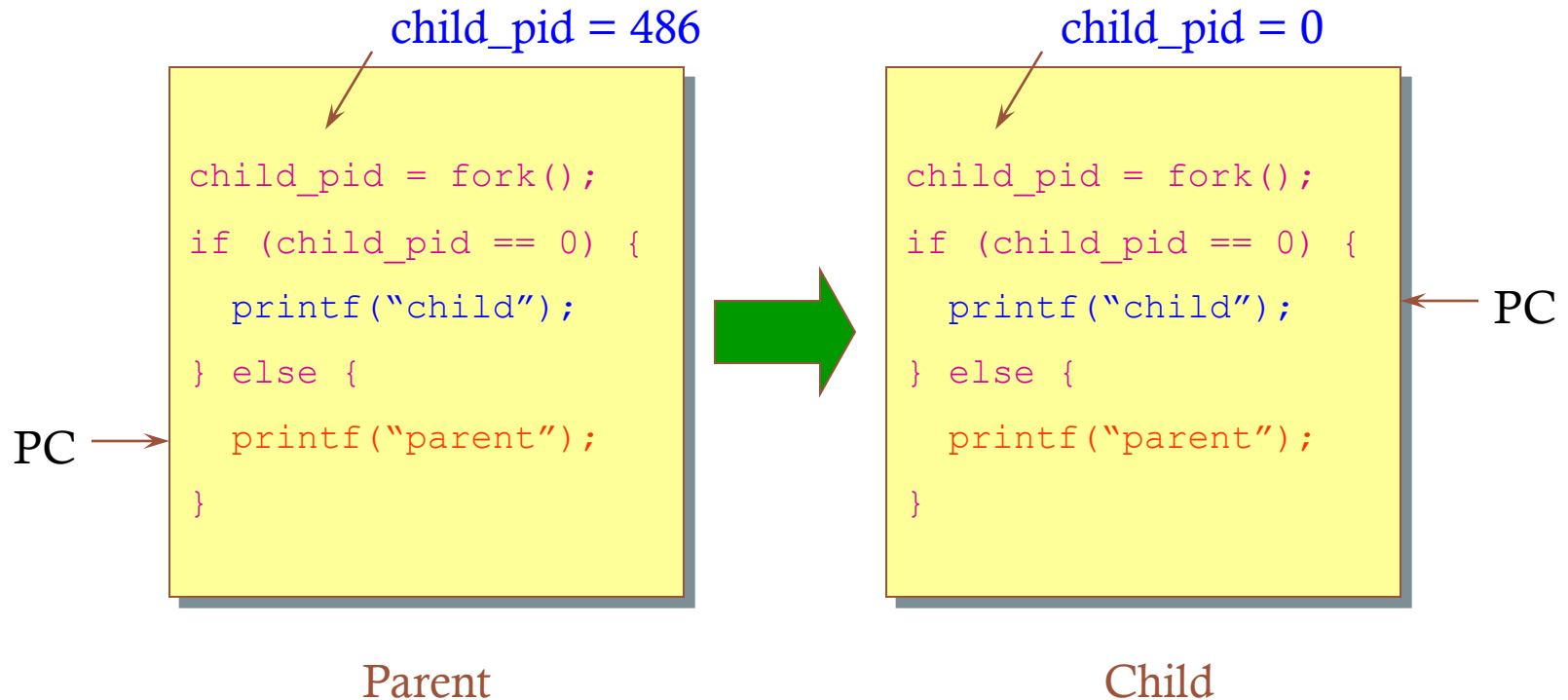
Child of a.out is 486



# Duplicating Address Spaces



# Divergence



# Example Continued

> 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  
    }  
}
```

# Process Creation: Unix (2)

- 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**
- What does it mean for `exec` to return?
- What does it mean for `exec` to return with an error?

# Process Creation: Unix (3)

- `fork()` is used to create a new process, `exec` is used to load a program into the address space
  - Why does Windows have `CreateProcess` while Unix uses `fork/exec`?
    - Comparing `fork()` and `CreateProcess()`?
    - Which is more convenient to use?
    - Which is more efficient?
- 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?
  - Cannot create child process (return to parent).

# 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
  - 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
- Note that a process does not **need** to clean up itself
  - Why does the OS have to do it?

# Process Termination

- When `exit()` is called on Unix:
  - Threads are terminated (next lec.)
  - Open files, network connections are closed
  - Address space is de-allocated
  - But the PCB still remains in the Process Table
- Only a parent can remove the PCB
  - Thus completely terminate the process (called **reap**)
- Died but not yet reaped process is called a **zombie**





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

# 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);  
    }  
}
```

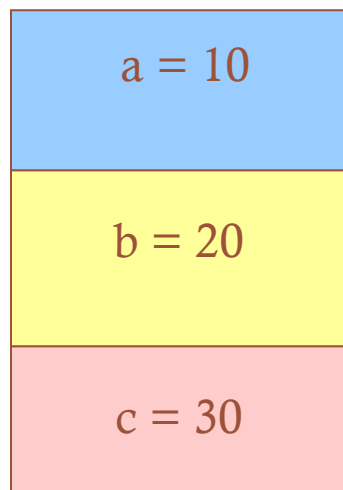
# Process Summary

- 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 (Windows), fork/exec (Unix)

# Copy-On-Write

- Lazy* copy

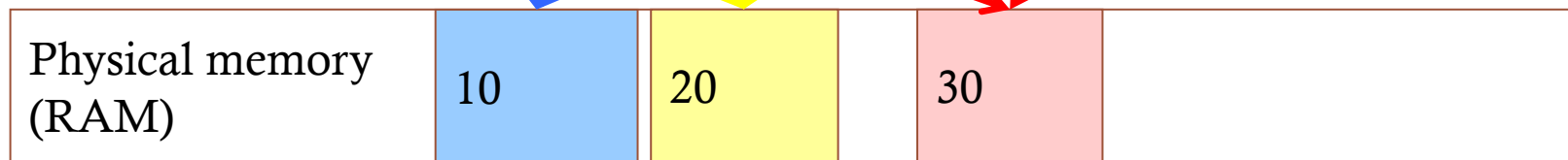
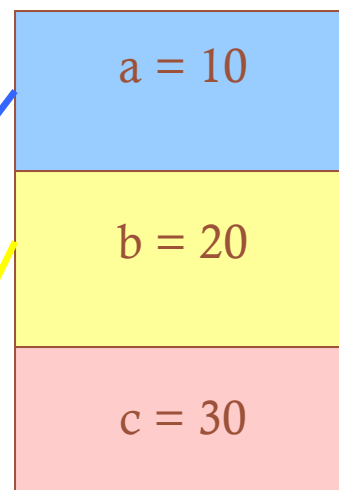
Proc. A's address space



`fork()`



Proc. B's address space



# Copy-On-Write

- Lazy* copy

