CE 440 Introduction to Operating System

Lecture 3: Processes
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Recap: Architecture Support for OS

Provide protection

- 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

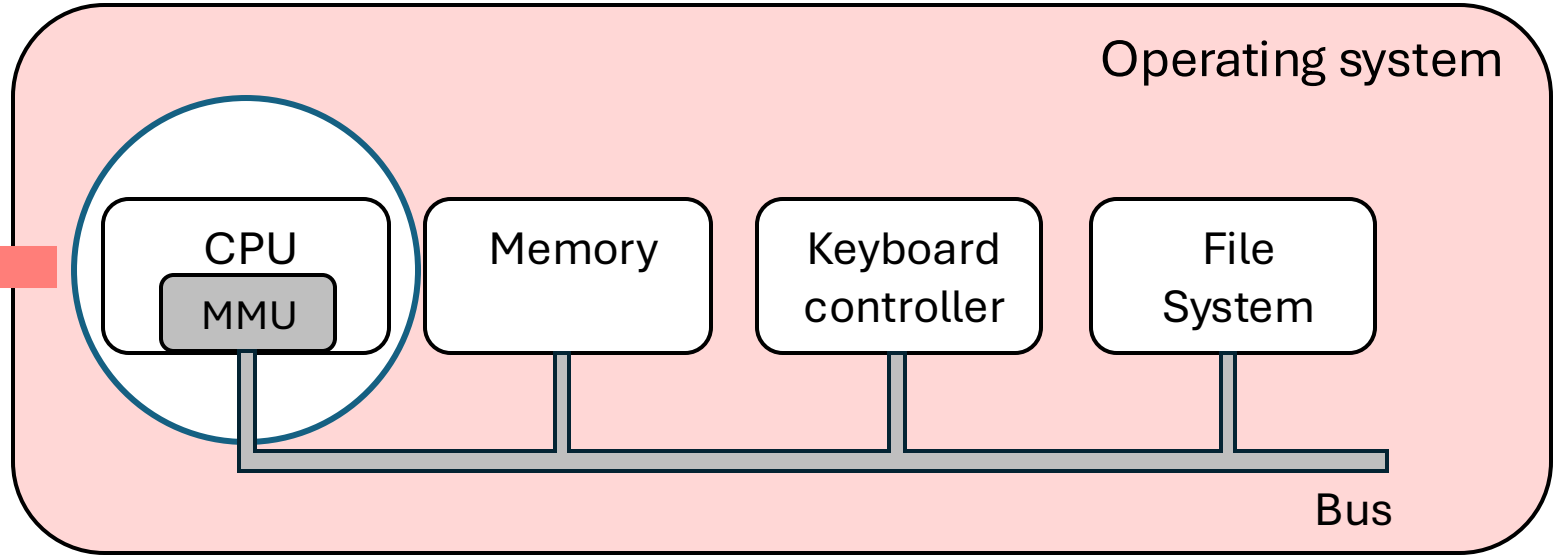
- Interrupts, atomic instructions

Today's Topic

Today's topic are processes and process management

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

OS abstraction?



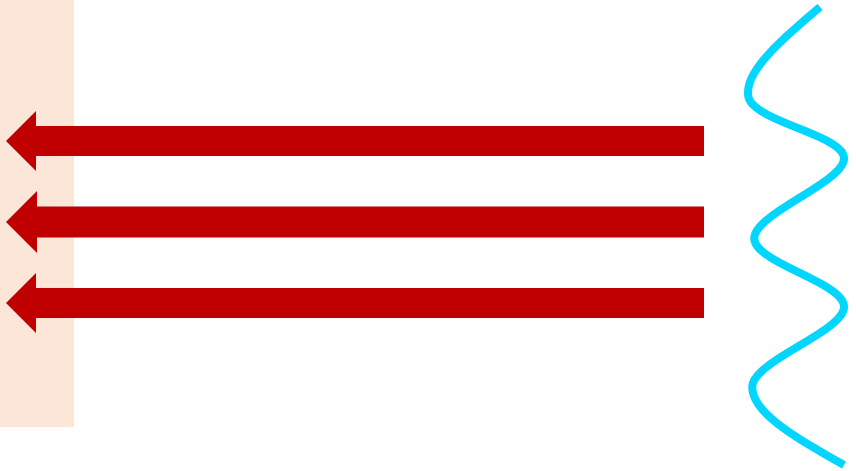
Process Abstraction

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

Processes vs. Program

```
int main() {
    int i, prod = 1;
    for (i=0;i<100;i++)
        prod = prod * i;
}
```



Program

- Static object existing in a file
- A sequence of instruction
- Static existence in space & time
- Same program can be executed by
- different processes

Process

- Dynamic object – program in execution
- A sequence of instruction executions
- Exists in limited span of time
- Same process may execute different program

Process Abstraction

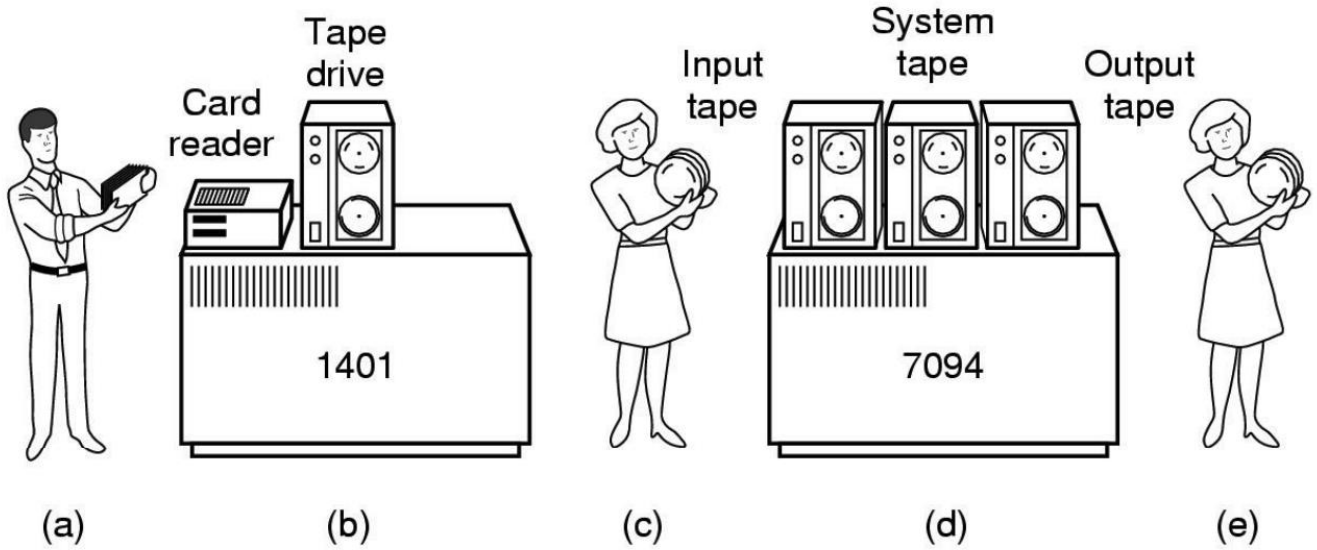
The process is the OS abstraction for CPU (execution)

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

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

Single Process: One-at-a-time



circa 1960s

Simple Process Management

Uniprogramming: a process runs from start to full completion

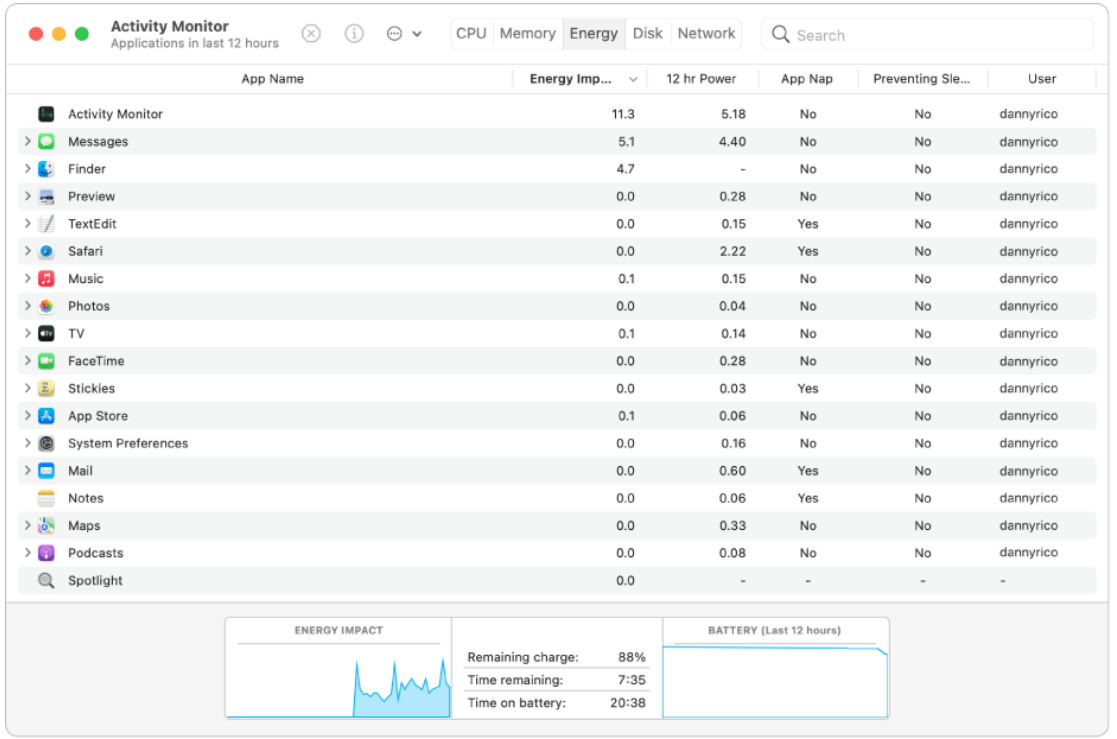
- What the early batch operating system does
- Load a job from disk (tape) into memory, execute it, unload the job
- Problem: low utilization of hardware
  - an I/O-intensive process would spend most of its time waiting for punched cards to be read
  - CPU is wasted
  - computers were very expensive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

Multiple Processes

Modern Oses run multiple processes simultaneously

```
last pid: 80032;  load averages:  0.84,  1.06,  1.05   up 199+03:28:06 22:26:35
107 processes: 1 running, 106 sleeping
CPU:  0.2% user,  5.0% nice,  1.0% system,  0.0% interrupt, 93.7% idle
Mem: 7938M Active, 3027M Inact, 2359M Wired, 683M Cache, 1643M Buf, 1842M Free
Swap: 8192M Total, 821M Used, 7371M Free, 10% Inuse

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU COMMAND
79957 www           1  52    8   356M 44704K accept  6   0:02 13.87% php-cgi
79477 www           1  52    8   360M 62256K accept  3   0:30  9.18% php-cgi
79471 www           1  52    8   356M 47792K accept  7   0:03  9.18% php-cgi
79476 www           1  34    8   356M 56140K accept  1   0:14  5.96% php-cgi
79933 www           1  40    8   356M 47340K accept  2   0:02  5.96% php-cgi
79958 pgsql         1  21    0  6399M   164M sbwait  0   0:00  2.10% postgres
79490 pgsql         1  21    0  6413M   652M sbwait  4   0:14  1.76% postgres
79474 www           1  28    8   360M 58620K sbwait  0   0:21  1.17% php-cgi
79475 www           1  30    8   360M 49396K accept  4   0:04  1.07% php-cgi
79934 pgsql         1  20    0  6407M   187M sbwait  0   0:00  0.98% postgres
79480 pgsql         1  20    0  6407M   397M sbwait  2   0:02  0.88% postgres
79482 pgsql         1  22    0  6403M   215M sbwait  3   0:01  0.88% postgres
79483 pgsql         1  20    0  6405M   495M sbwait  7   0:05  0.78% postgres
79470 www           1  30    8   356M 47476K accept  0   0:04  0.29% php-cgi
 1321 nobody        4  52    0   152M 85708K uwait   4  55.3H  0.10% memcached
79472 www           1  30    8   356M 46868K accept  6   0:02  0.10% php-cgi
 1308 pgsql         1  20    0  6391M  5328M select  1 287:40  0.00% postgres
```



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

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

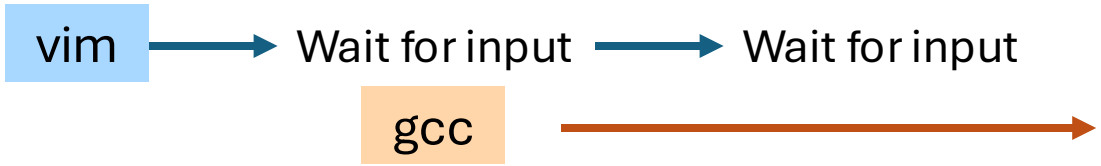
Why multiple processes (multiprogramming)?

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

Increased Utilization

Multiple processes can increase CPU utilization

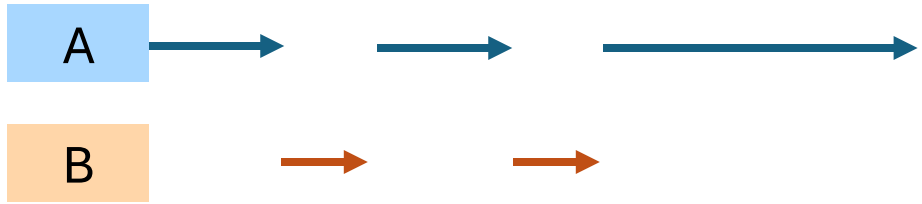
- Overlap one process's computation with another's wait



Multiple processes can reduce latency

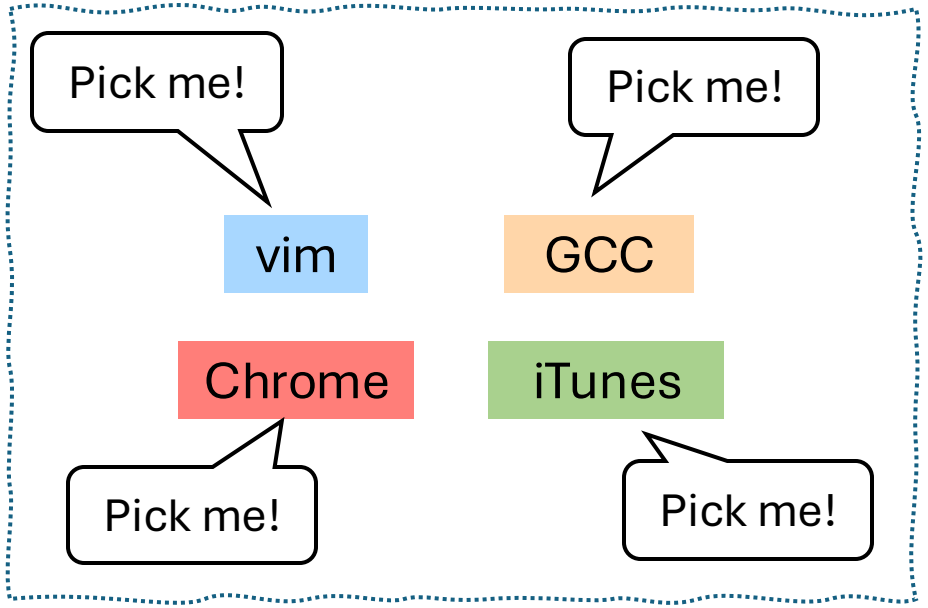
- Running A then B requires 100 second for B to complete

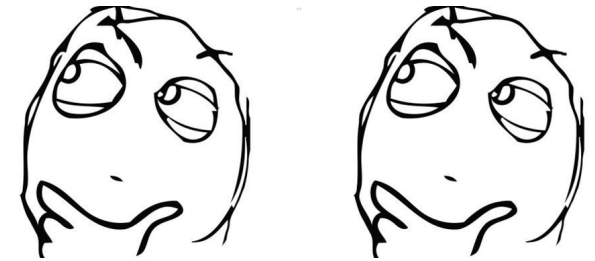
- Running A then B concurrently makes B finish faster



- < 100 second if both A and B are not completely CPU-bound

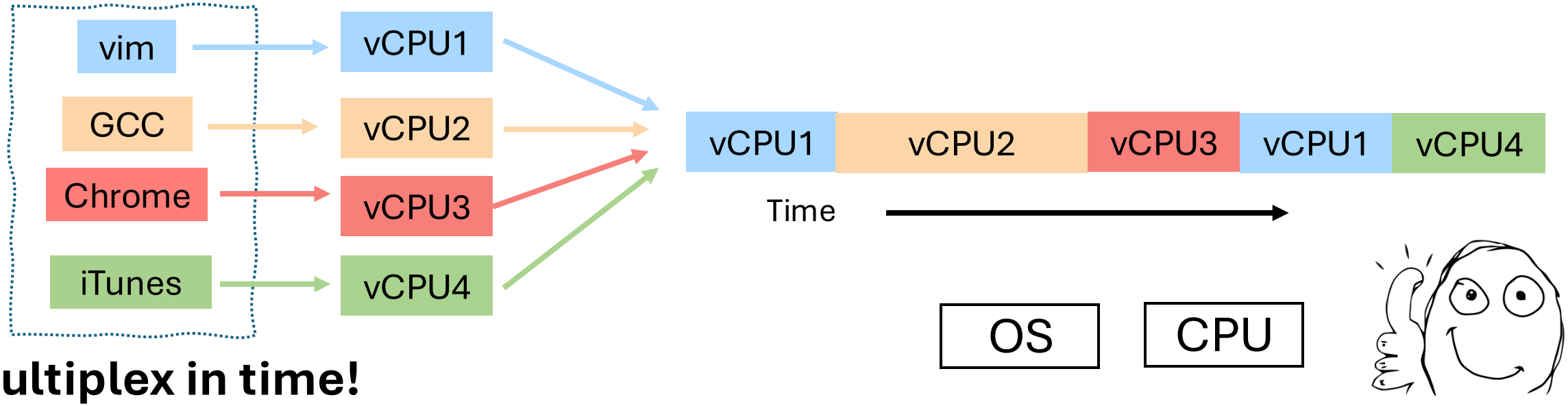
How to Implement Multiple Processors?





OS CPU

How to Implement Multiple Processors?



Multiplex in time!

Each virtual "CPU" needs a structure to hold:

- Program Counter (PC), Stack Pointer (SP)
- Registers

How switch from one virtual CPU to the next?

- Save PC, SP, and registers in current state block
- Load PC, SP, and registers from new state block

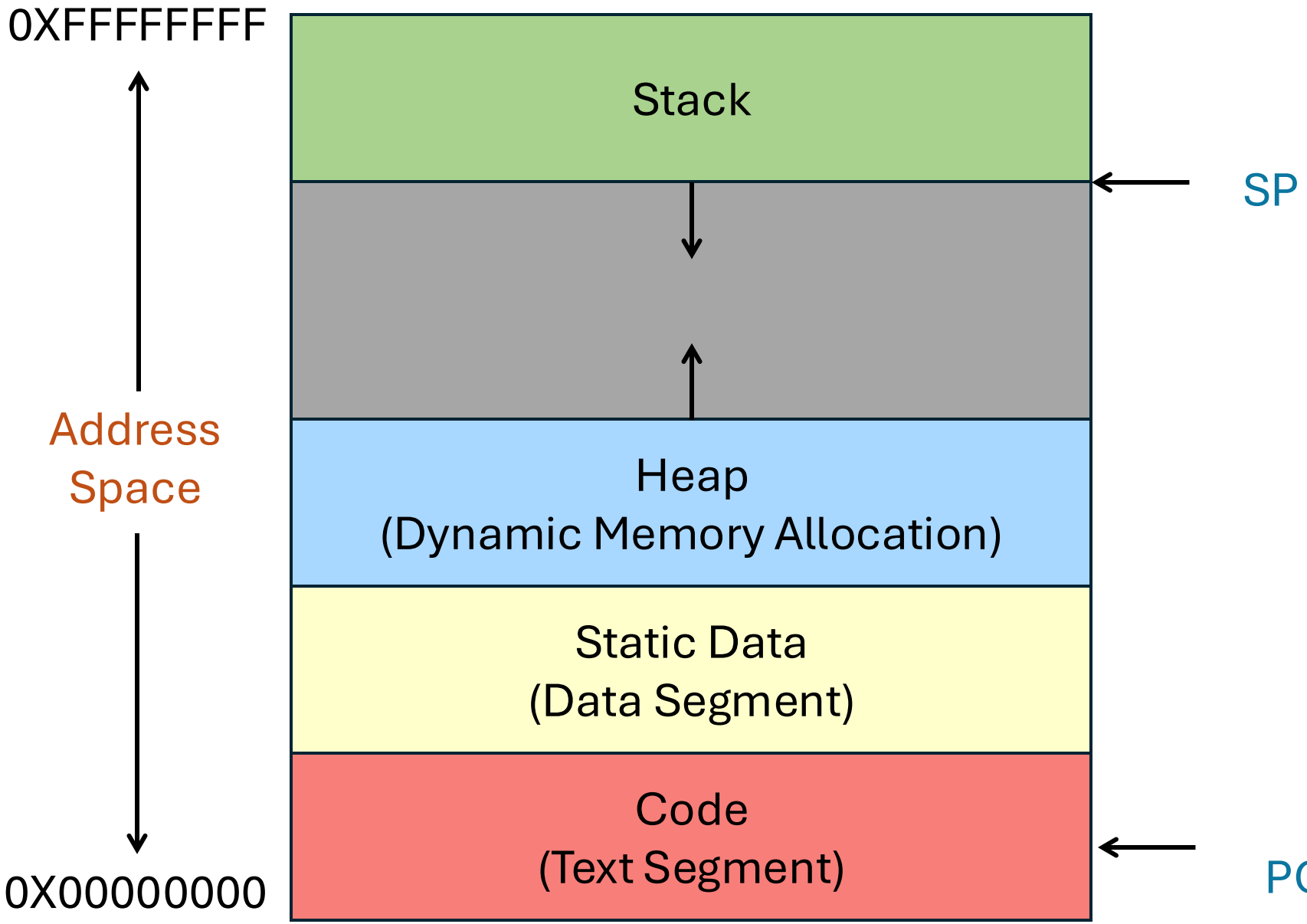
Processes in Kernel’s View

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

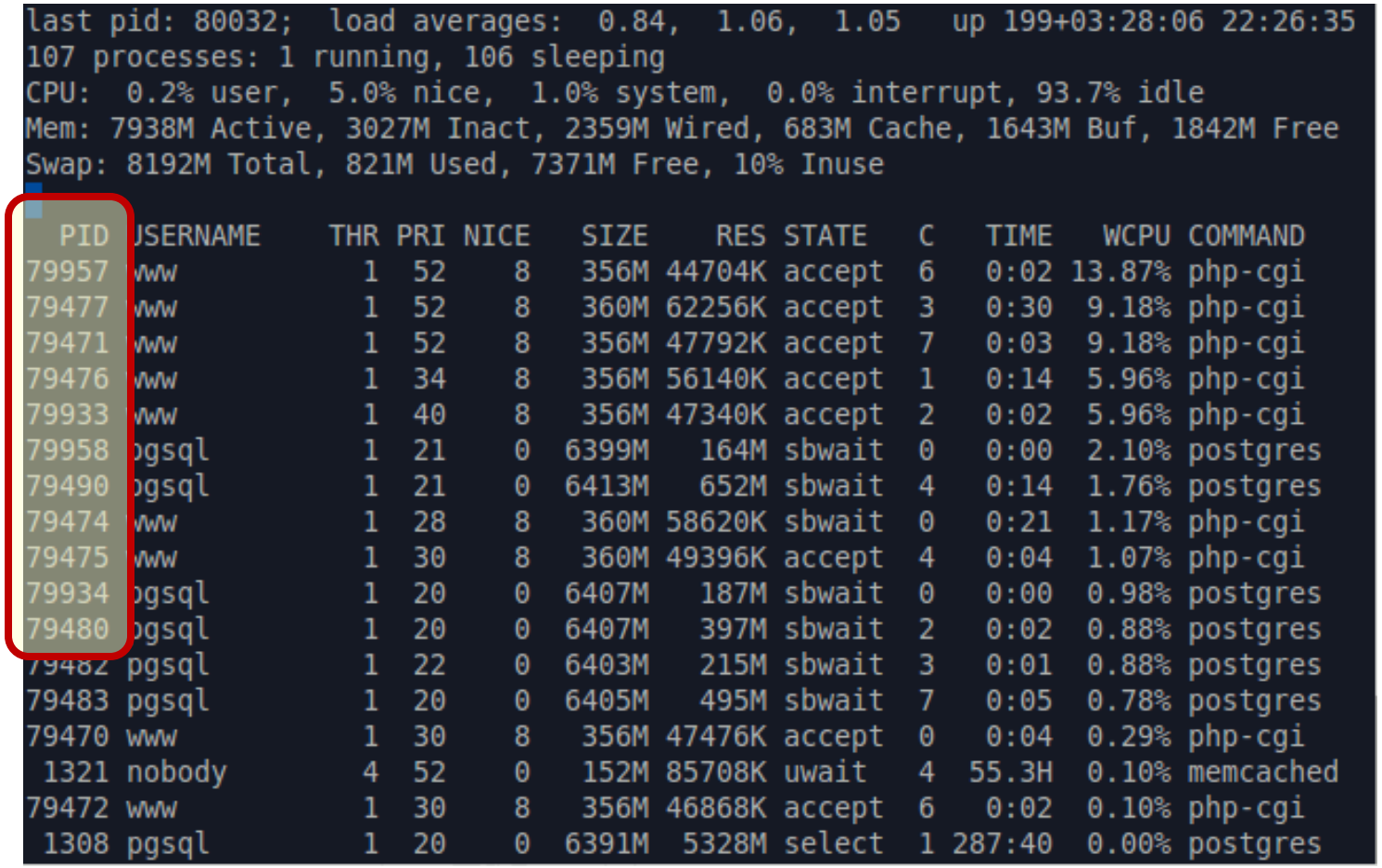
`*(char *)0xc000` **means different thing in P1 & P2**

Simplifies programming model

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

Naming A Process

The process is named using its process ID (PID)



```
last pid: 80032;  load averages:  0.84,  1.06,  1.05   up 199+03:28:06 22:26:35
107 processes: 1 running, 106 sleeping
CPU:  0.2% user,  5.0% nice,  1.0% system,  0.0% interrupt, 93.7% idle
Mem: 7938M Active, 3027M Inact, 2359M Wired, 683M Cache, 1643M Buf, 1842M Free
Swap: 8192M Total, 821M Used, 7371M Free, 10% Inuse

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU COMMAND
79957 www           1  52    8   356M 44704K accept  6   0:02 13.87% php-cgi
79477 www           1  52    8   360M 62256K accept  3   0:30  9.18% php-cgi
79471 www           1  52    8   356M 47792K accept  7   0:03  9.18% php-cgi
79476 www           1  34    8   356M 56140K accept  1   0:14  5.96% php-cgi
79933 www           1  40    8   356M 47340K accept  2   0:02  5.96% php-cgi
79958 pgsql         1  21    0  6399M   164M sbwait  0   0:00  2.10% postgres
79490 pgsql         1  21    0  6413M   652M sbwait  4   0:14  1.76% postgres
79474 www           1  28    8   360M 58620K sbwait  0   0:21  1.17% php-cgi
79475 www           1  30    8   360M 49396K accept  4   0:04  1.07% php-cgi
79934 pgsql         1  20    0  6407M   187M sbwait  0   0:00  0.98% postgres
79480 pgsql         1  20    0  6407M   397M sbwait  2   0:02  0.88% postgres
79482 pgsql         1  22    0  6403M   215M sbwait  3   0:01  0.88% postgres
79483 pgsql         1  20    0  6405M   495M sbwait  7   0:05  0.78% postgres
79470 www           1  30    8   356M 47476K accept  0   0:04  0.29% php-cgi
 1321 nobody        4  52    0   152M 85708K uwait   4  55.3H  0.10% memcached
79472 www           1  30    8   356M 46868K accept  6   0:02  0.10% php-cgi
 1308 pgsql         1  20    0  6391M  5328M select  1 287:40  0.00% postgres
```

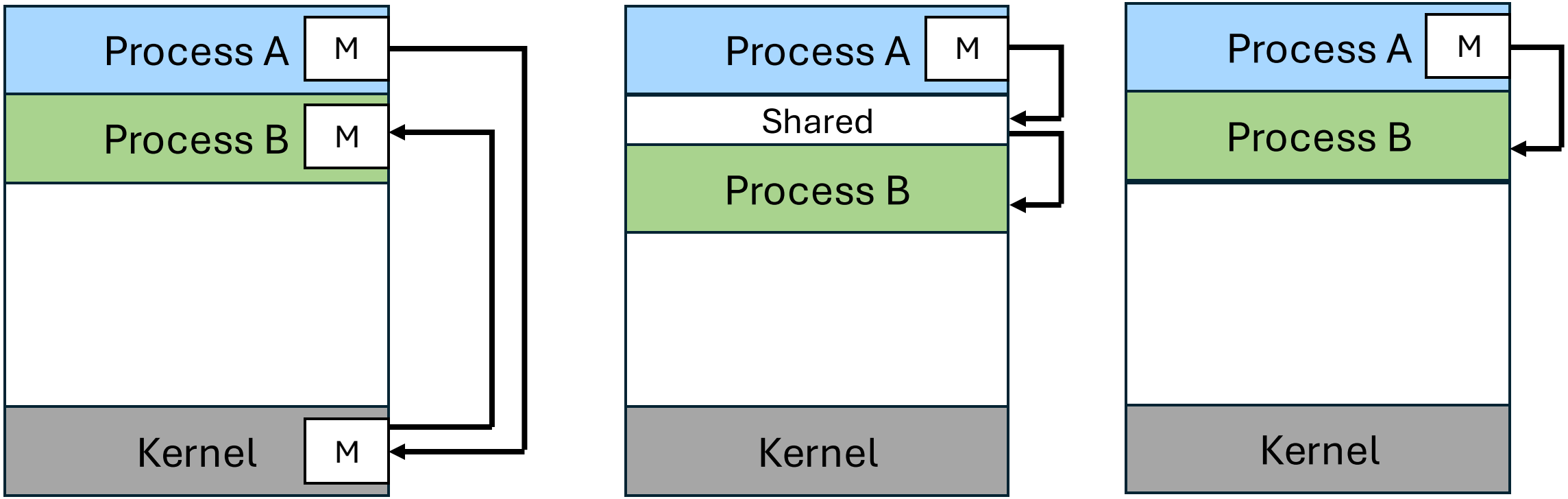
Inter-Process Communication (IPC)

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

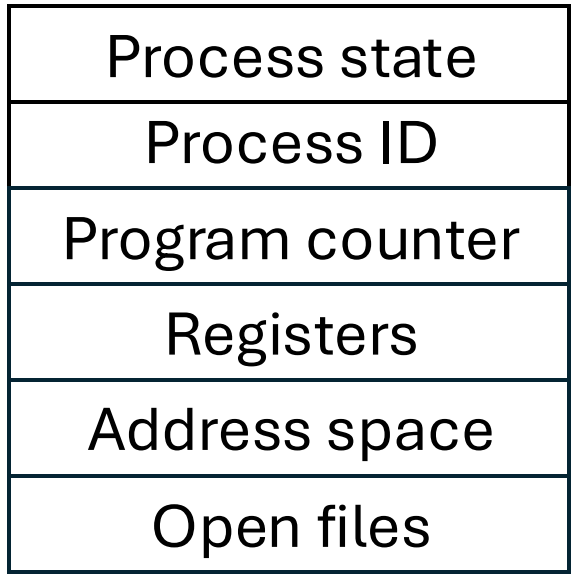
Implementing Process

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

Implementing Process

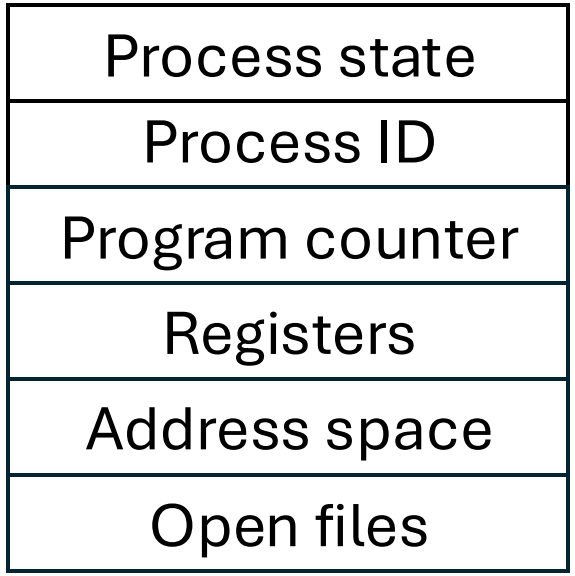
PCB includes information necessary for execution

- 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!



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;
        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 */
```

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" */
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 */
f
    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

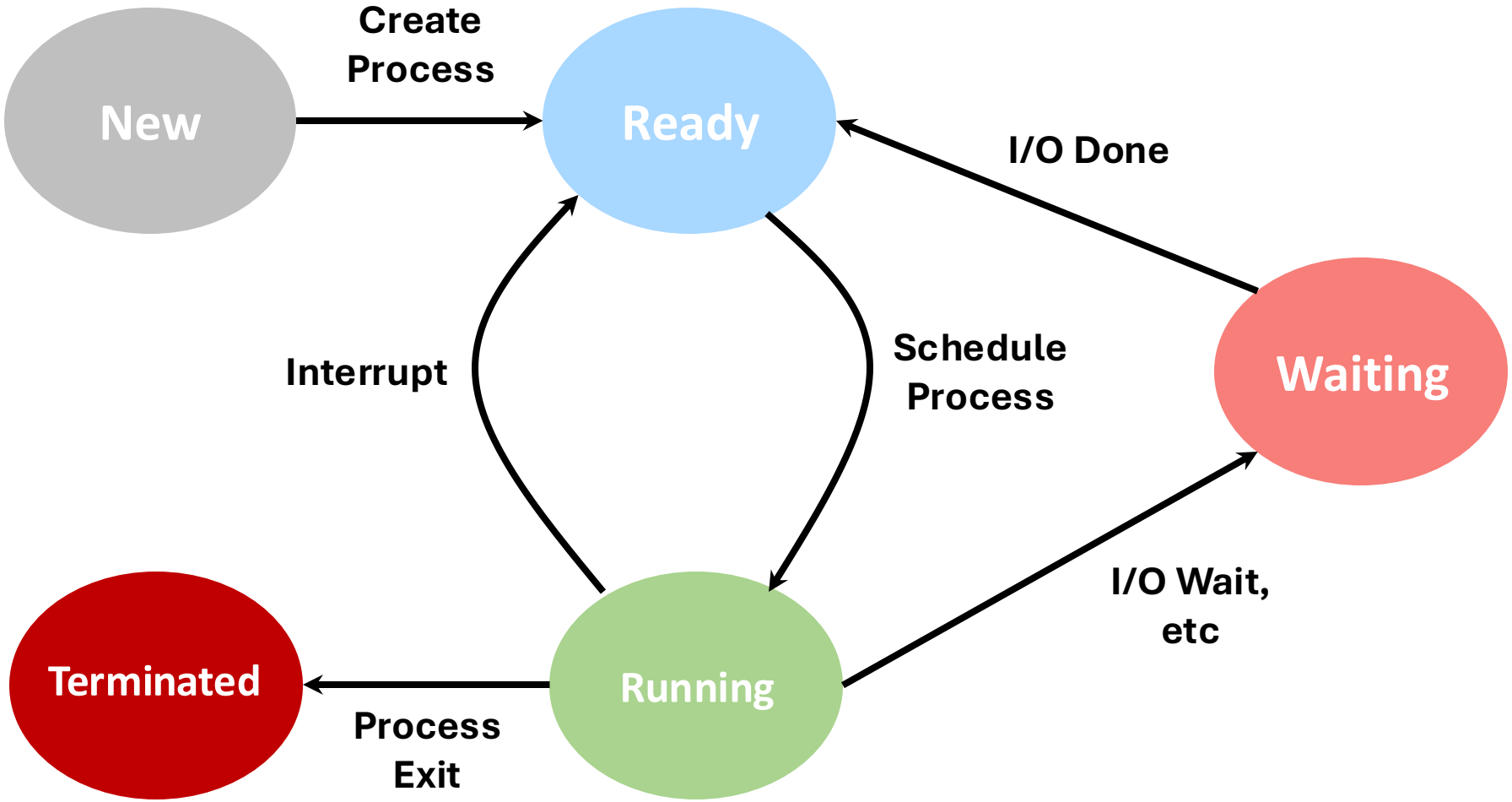
- It cannot make progress until event is signaled (disk completes)

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?

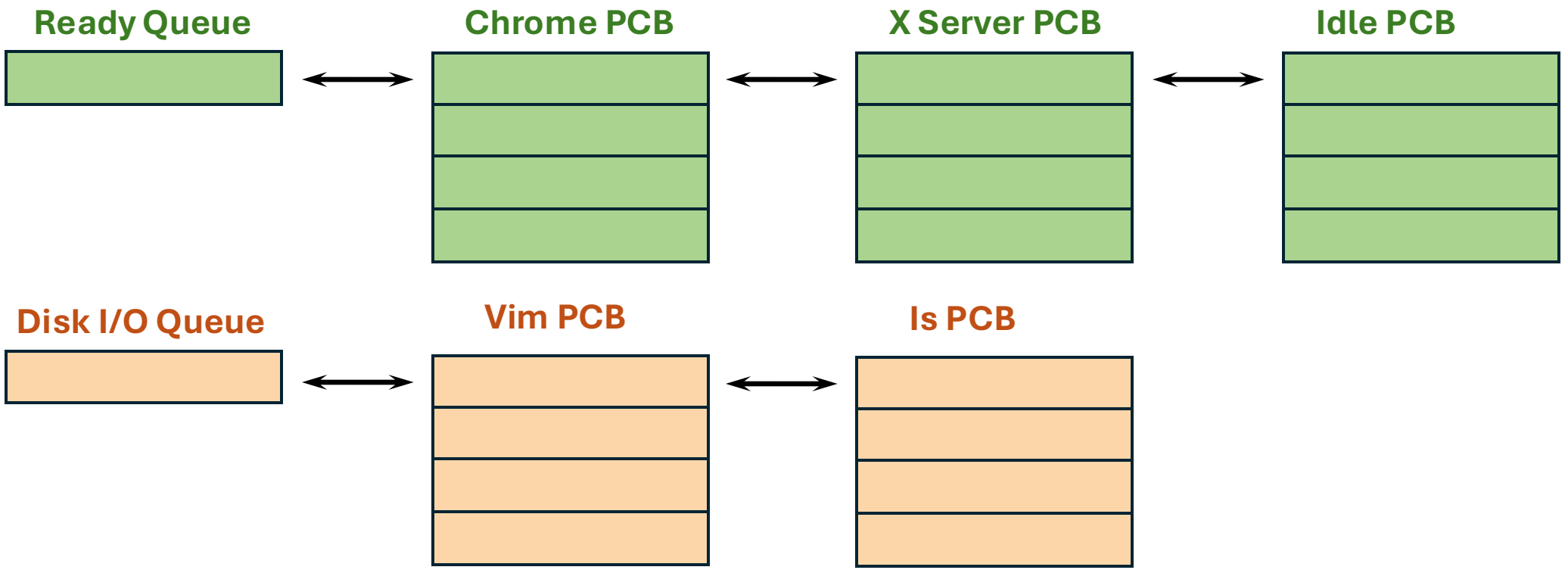
Naïve approach: process list

- 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



Console Queue

Sleep Queue

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

Question ?

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

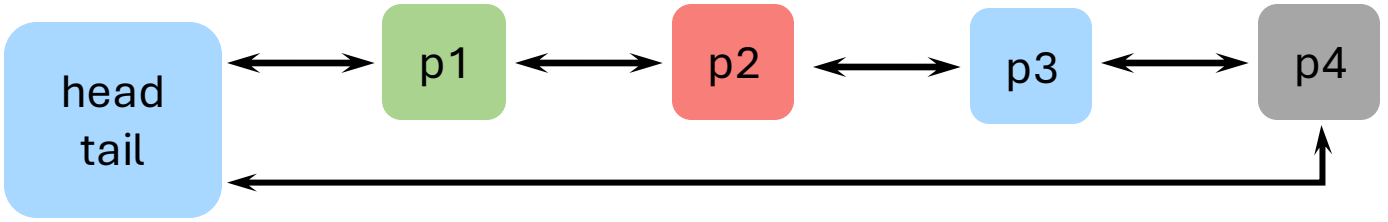
Scan process table for first runnable?

- Expensive. Unfairness (small pids do better)

Better Scheduling

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

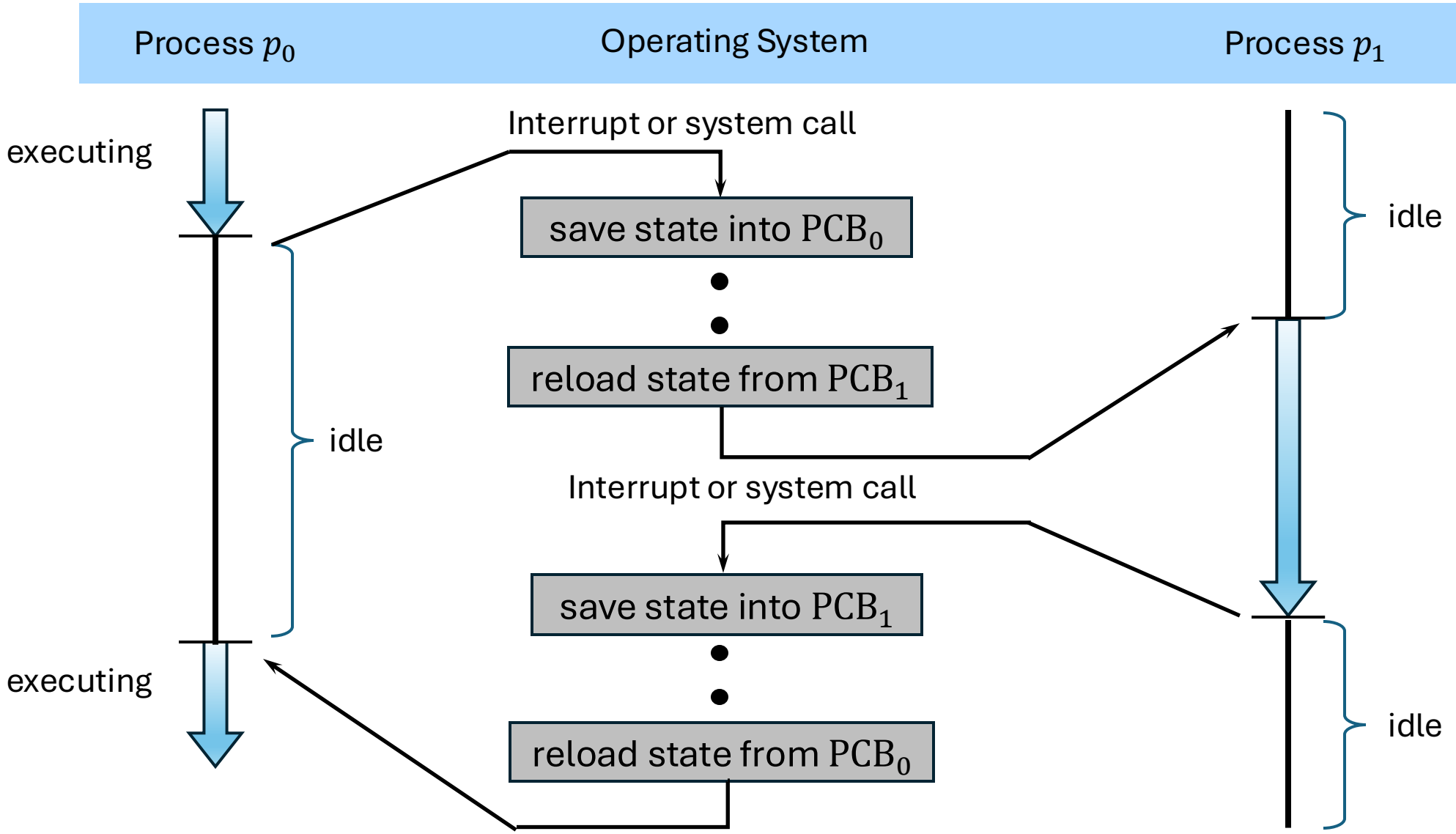
- Disk request completed, or packet arrived on network
- Previously waiting process becomes runnable

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)

Question ?

How to use 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

Creating Process in Windows

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

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

`CreateProcess:`

1. Create 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
6. specified)
7. Places the PCB on the ready queue

CreateProcessA function (processthreadsapi.h)

02/08/2023

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 CreateProcessAsUserA function or CreateProcessWithLogonW function.

Syntax

C++

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Creating Process in Unix

In Unix, processes are created using fork()

```
int 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

NAME top

fork – create a child process

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION top

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).
- Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.

Fork()

```c
#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 the program prints?

Example Output

```c
#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 the program prints?

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 are processes created?

- CreateProcess (NT), fork/exec (Unix)

Next time…

Read Chapters 26, 27