Modern Operating Systems

Chapter 2 – Process&Thread

Zhang Yang

Spring 2022

Content of this lecture

- 2.1 Processes
- 2.2 Threads

- 2.1 Processes
- 2.2 Threads

Modern Workstations and PCs

- Multiple windows in personal computer doing completely independent things.
  - Word
  - Excel
  - Photoshop
  - E-mail
  - Music player, etc.

Multiprogramming (1)

- When there is a single program running in the CPU, it leads to the degradation of the CPU utilization.
  - Example: When a running program initiates an I/O operation, the CPU remain idle until the I/O operation is completed.

(a) Uniprogramming

  - Solution to this problem is provided by *Multiprogramming*.

Multiprogramming (2)

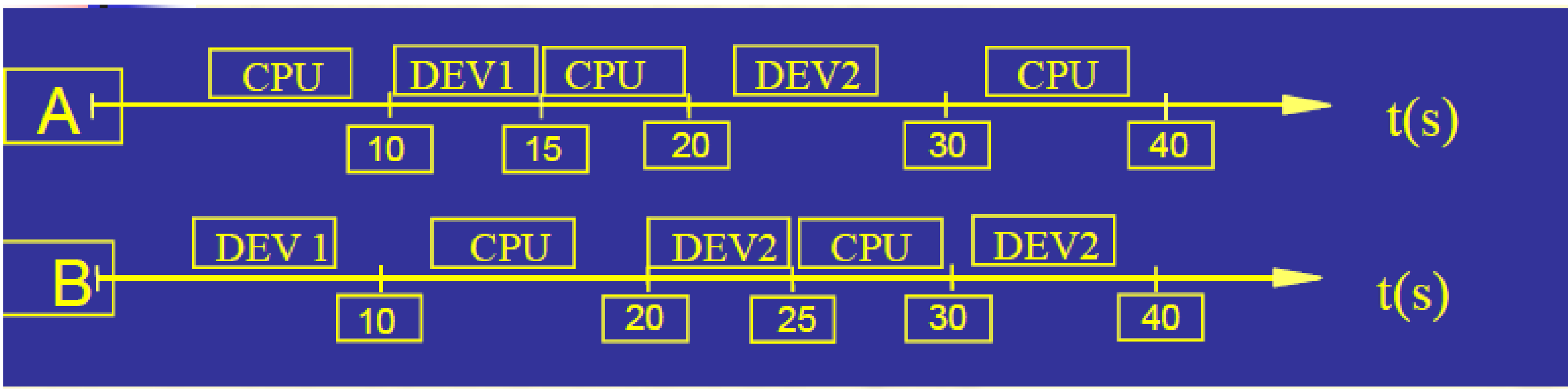
- What is Multiprogramming?
  - A mode of operation that provides for the **<u>interleaved execution</u>** of two or more programs by a single processor.
- Improving CPU utilization
  - By allowing several programs to reside in main memory at the "**<u>same time</u>**" the CPU might be shared, such that when one program initiates an I/O operation, another program can be assigned to the CPU, thus the improvement of the CPU utilization.

Multiprogramming (3)

- Degree of Multiprogramming
  - The number of processes loaded simultaneously in memory is called degree of multiprogramming.

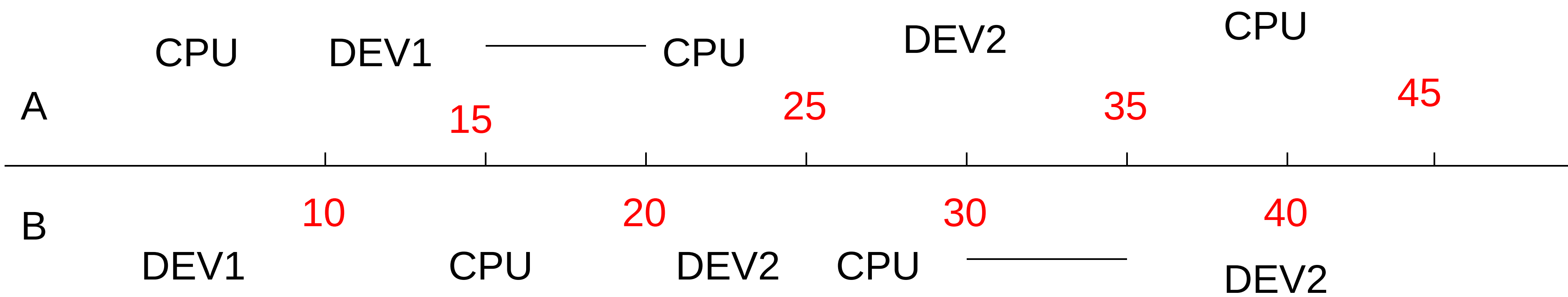
Concurrency (1)

- Allows multiple applications to run “at the same time”.
- Example
  - In sequential environment, A executes first, then B.



Concurrency (2)

- Example (ctd.)
  - Concurrency



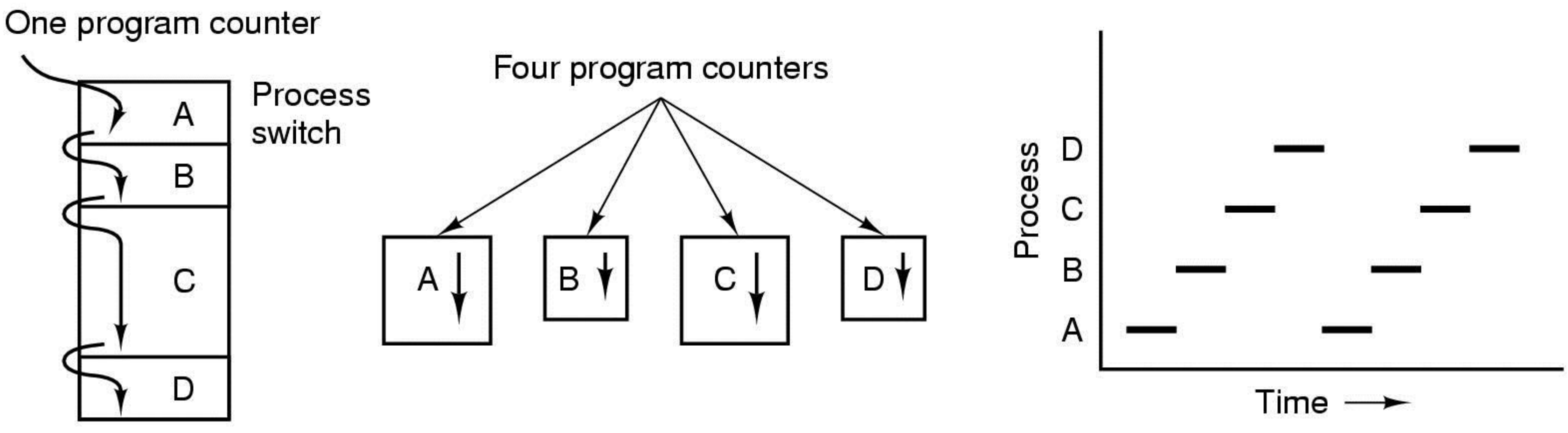
Concurrency (3)

- Benefits of Concurrency
  - What are the limitations without concurrency?
    - Long response time
    - Resources under utilized
  - Better resource utilization
    - Resources unused by one application can be used by the others.
  - Better average response time
    - No need to wait for other applications to complete.

The Process Model (1)

- The process is an OS abstraction for a running program.
- Process: an executing program
  - Program = static file (image)
  - Process = executing program = program + execution state.
- Processes are instances of a program.
  - Different processes may run different instances of the same program.
    - E.g., My javac and your javac process both run the Java compiler.

The Process Model (2)



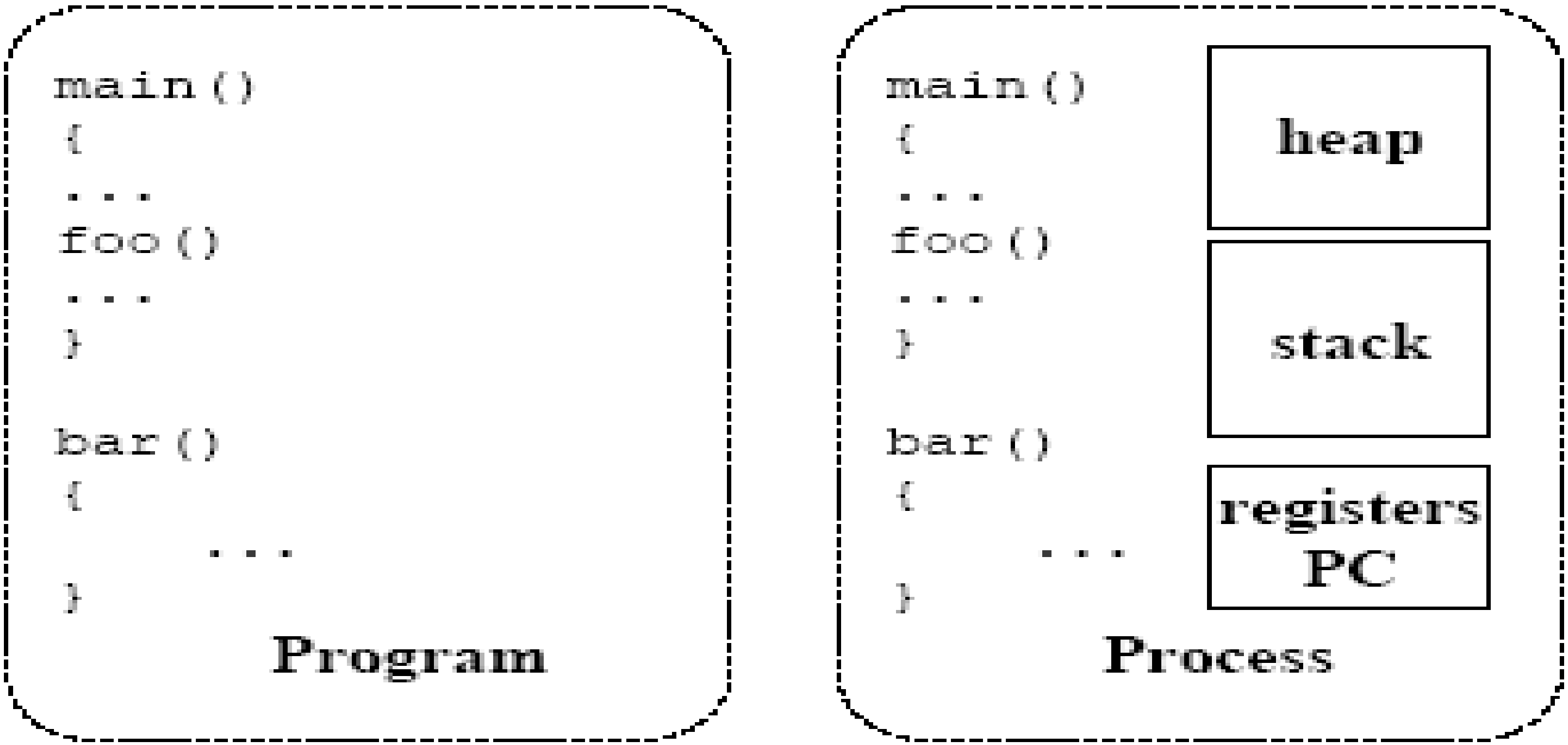
(a) (b) (c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
- Real life analogy?

Program and Process (1)

- Analogy
  - Program: steps for attending the lecture
    - Step1: walk to A1
    - Step2: enter 409
    - Step3: find a seat
    - Step4: listen and take notes (or sleep)
  - Process: attending the lecture
    - Action
    - You are all in the middle of a process

Program and Process (2)



Program and Process (3)

- Differences between program and process

	Program	Process
1	Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
2	Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.
3	Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
4	Program is a static entity.	Process is a dynamic entity.

Program and Process (4)

- Differences between program and process (ctd.)

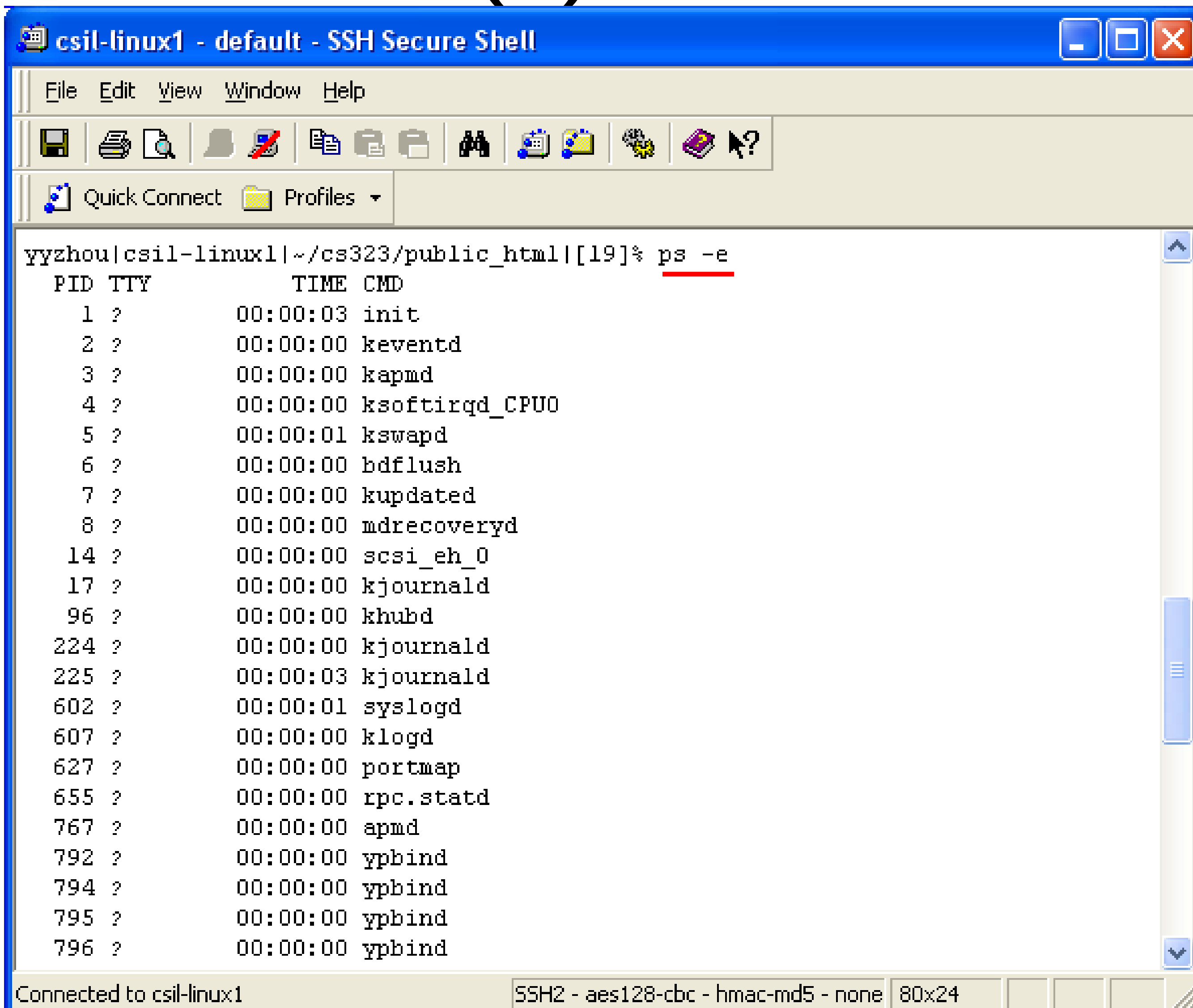
	Program	Process
5	Program does not have any resource requirement, it only requires memory space for storing the instructions.	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime.
6	Program does not have any control block.	Process has its own control block called Process Control Block.
7	Program has two logical components: code and data.	In addition to program data, a process also requires additional information required for the management and execution.
8	Program does not change itself.	Many processes may execute a single program. Their program code may be the same but program data may be different. These are never same.

Process Creation (1)

- Process Creation Events
  - System Initialization
    - Reboot
      - Foreground process
      - Background process, daemon process
  - Execution of a process creation system call
    - fork()
  - User request to create a new process
    - Command line or click an icon
  - Initiation of a batch job

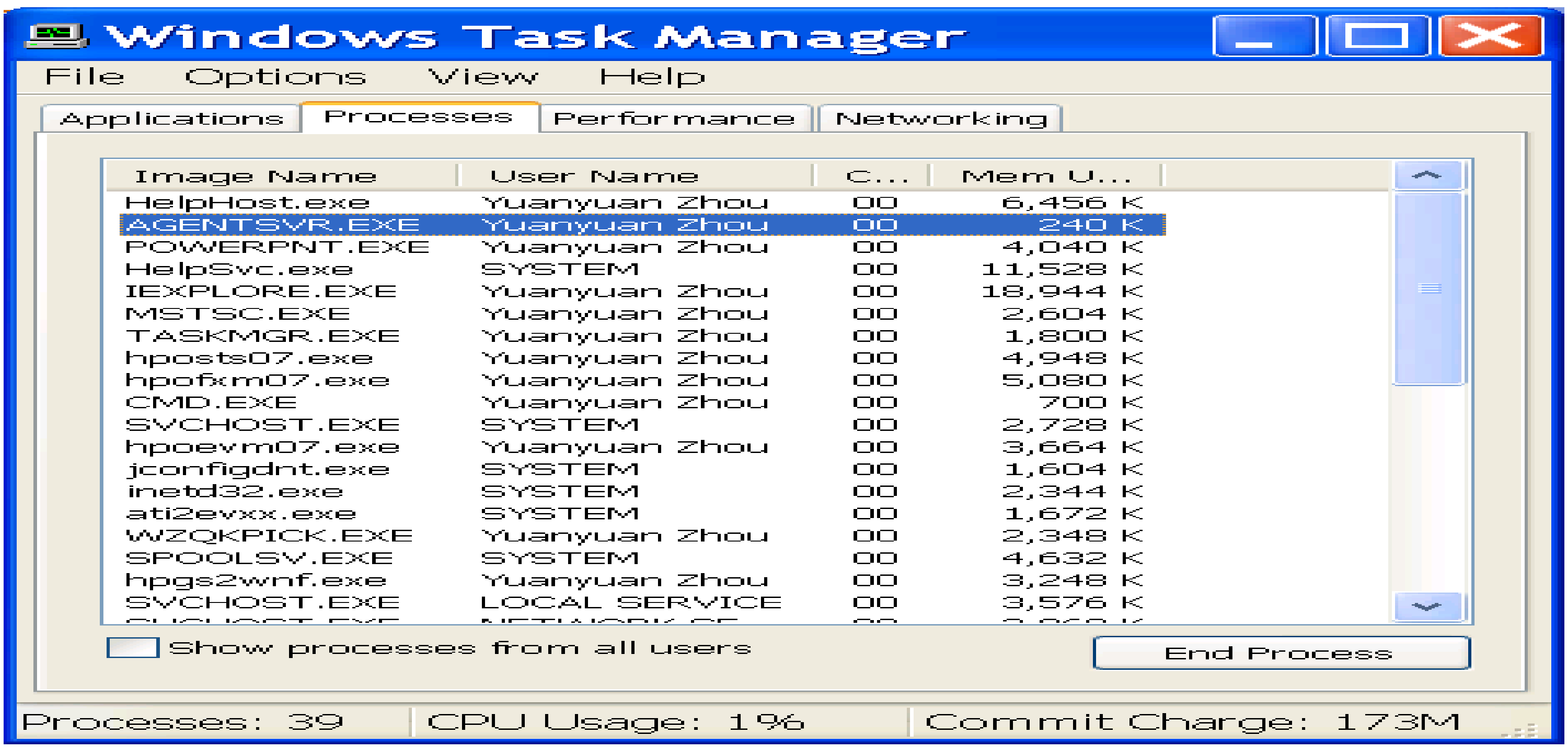
Process Creation (2)

- Unix: ps



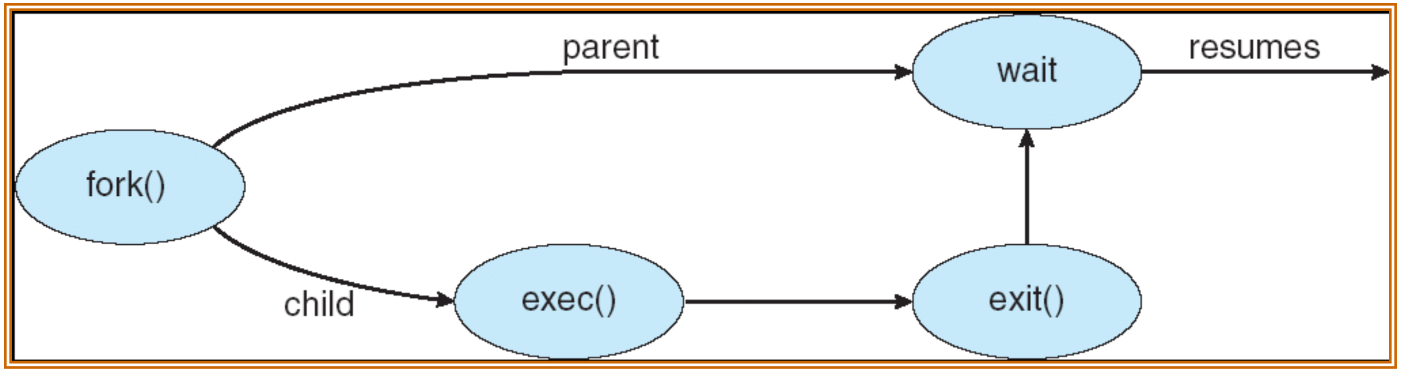
Process Creation (3)

- Windows: Task Manager



Process Creation (4)

- UNIX Example
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process’ memory space with a new program



Process Creation (5)

- UNIX Example (ctd.)
  - Before fork

PC

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("Process %d: value is %d\n", getpid(), i);

        if (i == 5) {
            printf("Process %d: About to do a fork...\n", getpid());
            int child_pid = fork();
        }
    }

}
```

Process Creation (6)

- UNIX Example (ctd.)
  - After fork

PC

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < 10; i++) {
      printf("Process %d: value is %d\n", getpid(), i);

      if (i == 5) {
        printf("Process %d: About to do a fork...\n", getpid());
        int child_pid = fork();
      }
    }

}
```

Parent Process

PC

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < 10; i++) {
      printf("Process %d: value is %d\n", getpid(), i);

      if (i == 5) {
        printf("Process %d: About to do a fork...\n", getpid());
        int child_pid = fork();
      }
    }

}
```

Child Process

Process Creation (7)

- UNIX Example (ctd.)
  - Output of sample program
    - Process 4530: value is 0
    - Process 4530: value is 1
    - Process 4530: value is 2
    - Process 4530: value is 3
    - Process 4530: value is 4
    - Process 4530: value is 5
    - Process 4530: About to do a fork...
    - **Process 4531: value is 6**
    - Process 4530: value is 6
    - Process 4530: value is 7
    - **Process 4531: value is 7**
    - Process 4530: value is 8
    - **Process 4531: value is 8**
    - Process 4530: value is 9
    - **Process 4531: value is 9**
  - *What determines the order in which the two processes run???*

Process Creation (8)

- Windows Example

BOOL CreateProcess(

LPCTSTR *lpApplicationName*, // pointer to name of executable module

LPTSTR *lpCommandLine*, // pointer to command line string

LPSECURITY_ATTRIBUTES *lpProcessAttributes*, // process security attr.

LPSECURITY_ATTRIBUTES *lpThreadAttributes*, // thread security attr.

BOOL *bInheritHandles*, // handle inheritance flag

DWORD *dwCreationFlags*, // creation flags

LPVOID *lpEnvironment*, // pointer to new environment block

LPCTSTR *lpCurrentDirectory*, // pointer to current directory name

LPSTARTUPINFO *lpStartupInfo*, // pointer to STARTUPINFO

LPPROCESS_INFORMATION *lpProcessInformation* // pointer to

PROCESS_INFORMATION **)**

Process Termination

- Process Termination Events
  - Normal exit (voluntary)
    - Unix: exit
    - Windows: ExitProcess
  - Error exit (voluntary)
    - e.g., cc foo.c
  - Fatal error (involuntary)
    - Divide by 0, executing an illegal instruction, referencing nonexistent memory
  - Killed by another process (involuntary)
    - Kill , TerminateProcess

Process Hierarchies (1)

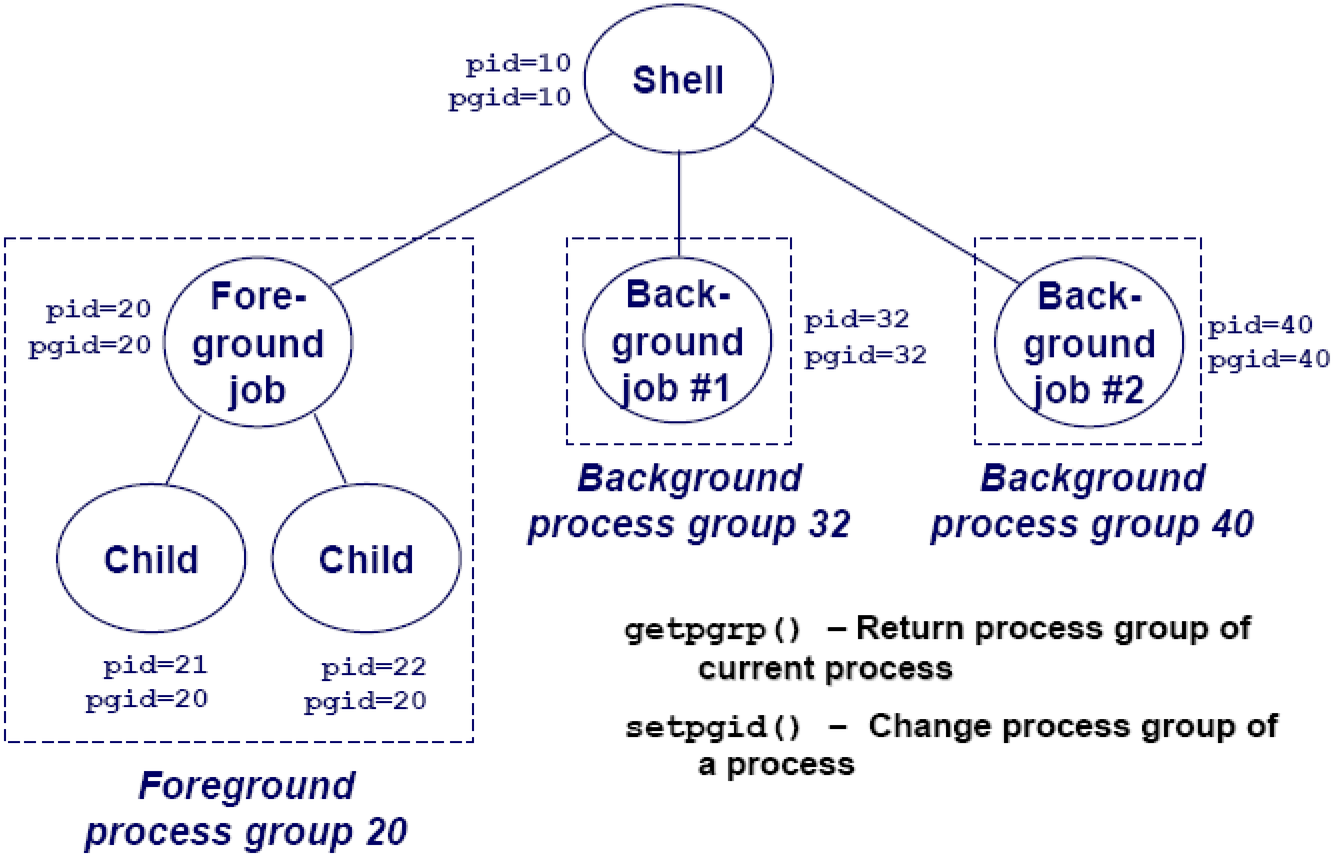
- Parent creates a child process, a child process can create its own processes.
- Forms a hierarchy
  - UNIX calls this a "process group“.
- Windows has no concept of process hierarchy
  - All processes are created equal.

Process Hierarchies (2)

- Process Groups
  - A process group is a collection of processes established for purposes such as signal delivery.
  - Each process has a process group ID that identifies the process group to which it belongs.
  - When a process generates child processes, the operating system automatically creates a process group.
  - The initial parent process is known as the process leader. The process leader's PID will be the same as its process group ID.
  - A process can find its process group ID from the system call ***getpgid*** .
  - A process may change its process group by using the system call ***setpgid***.

Process Hierarchies (3)

- Process Groups
  - Example

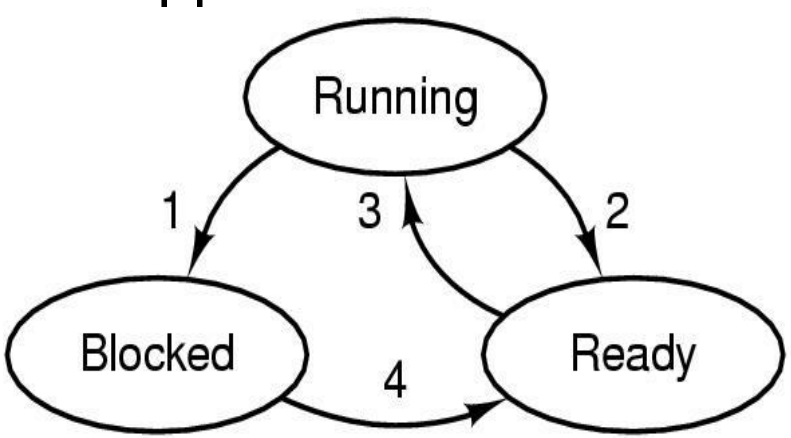


Process States (1)

- Process States
  - Running
    - Process is currently using the CPU.
  - Ready
    - Currently waiting to be assigned to a CPU.
    - That is, the process could be running, but another process is using the CPU.
  - Blocked
    - Process is waiting for an event.
      - Such as completion of an I/O, a timer to go off, a resource becoming available, etc.
    - Why is this different than “ready” ?

Process States (2)

- Transitions between states
  - Running—>Blocked
    - A process discover that it cannot continue.
  - Running —>Ready, Ready —>Running
    - Caused by process scheduler.
  - Blocked —>Ready
    - The external event for which a process was waiting happens.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Group Discussion (2 minutes)

- Can the following transitions occur?
  - Ready to blocked
  - Blocked to running

Process Control Block (1)

- Process Control Block (PCB, also called Task Control Block or Task Struct) is a data structure in the operating system kernel containing the information needed to manage a particular process.
- OS maintains a PCB for each process.
- A PCB keeps all the information needed to keep track of a process.
- The PCB is identified by an integer process ID.

Process Control Block (2)

- The PCB is maintained for a process throughout its lifetime and is deleted once the process terminates.
- The architecture of a PCB is completely dependent on operating system and may contain different information in different operating system.

Process Control Block (3)

- PCB contains
  - Process ID
    - Unique identification for each of the process in the operating system.
  - State
    - The current state of the process.
  - Pointer
    - A pointer to parent process.
  - Priority
    - Priority of a process.
  - Program Counter

Process Control Block (4)

- PCB contains (ctd.)
  - CPU Registers
    - Various CPU registers where process need to be stored for execution for running state.
  - I/O Info
    - I/O status information includes a list of I/O devices allocated to the process.
  - Accounting Info
    - This includes the amount of CPU used for process execution, time limits etc.

Process Control Block (5)

- An Example of PCB

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	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 Fields of a process table entry

Process Control Block (6)

- Linux PCB Structure: task_struct

```
struct task_struct {
volatile long state;          Execution state
unsigned long flags;
int sigpending;
mm_segment_t addr_limit;
struct exec_domain *exec_domain;
volatile long need_resched;
unsigned long ptrace;
int lock_depth;
unsigned int cpu;
int prio, static_prio;
struct list_head run_list;
prio_array_t *array;
unsigned long sleep_avg;
unsigned long last_run;
unsigned long policy;
unsigned long cpus_allowed;
unsigned int time_slice, first_time_slice;
atomic_t usage;
struct list_head tasks;
struct list_head ptrace_children;
struct list_head ptrace_list;
struct mm_struct *mm, *active_mm;          Memory mgmt info
struct linux_binfmt *binfmt;
int exit_code, exit_signal;
int pdeath_signal;
unsigned long personality;
int did_exec:1;
unsigned task_dumpable:1;
pid_t pid;          Process ID
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
int leader;
struct task_struct *real_parent;
struct task_struct *parent;
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
struct pid_link pids[PIDTYPE_MAX];
wait_queue_head_t wait_chldexit;
struct completion *vfork_done;
int *set_child_tid;
int *clear_child_tid;
unsigned long rt_priority;          Priority
```

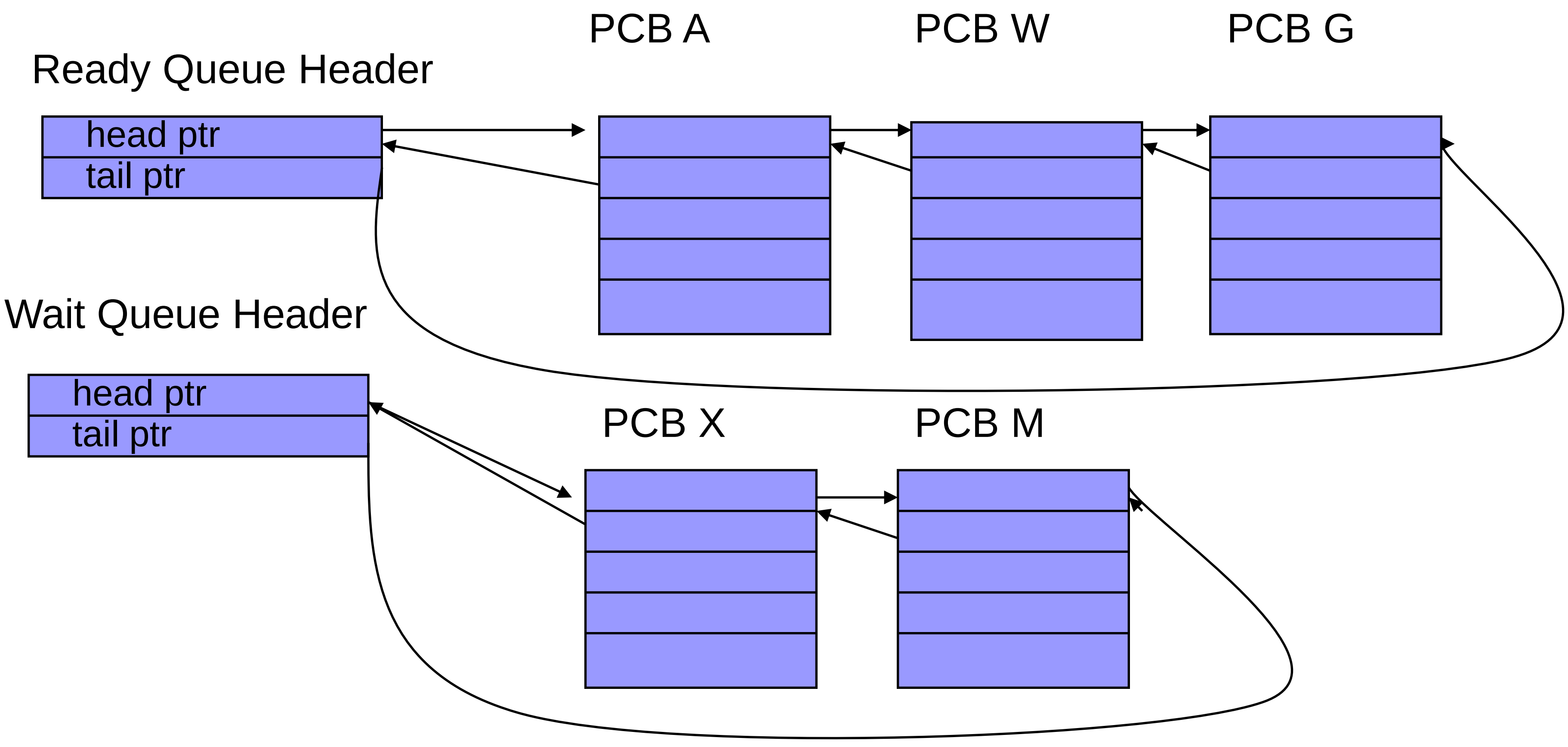
```
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;          Accounting info
struct tms group_times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt,
cnswap;
int swappable:1;
uid_t uid,euid,suid,fsuid;          User ID
gid_t gid,egid,sgid,fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
int link_count, total_link_count;
struct tty_struct *tty;
unsigned int locks;
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct thread_struct thread;          CPU state
struct fs_struct *fs;
struct files_struct *files;          Open files
struct namespace *namespace;
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
void *tux_info;
void (*tux_exit)(void);
      u32 parent_exec_id;
      u32 self_exec_id;
spinlock_t alloc_lock;
        spinlock_t switch_lock;
void *journal_info;
unsigned long ptrace_message;
```

Implementation of Processes (1)

- State Queues
  - The OS maintains a collection of queues that represent the state of all processes in the system.
  - There is typically one queue for each state, e.g., ready, waiting for I/O, etc.
  - Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is unlinked from one queue and linked onto another.

Implementation of Processes (2)

- State Queues (ctd.)



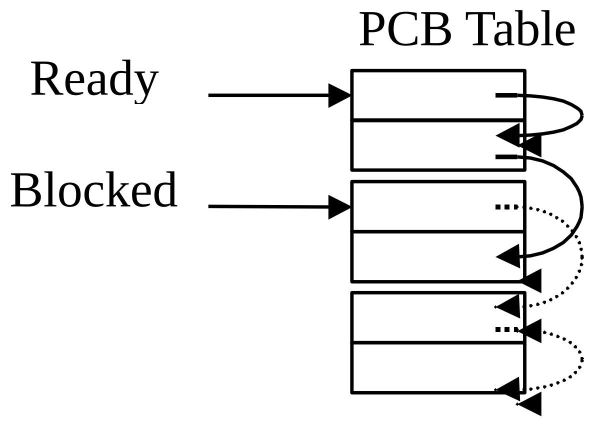
There may be many wait queues, one for each type of wait (specific device, timer, message,…).

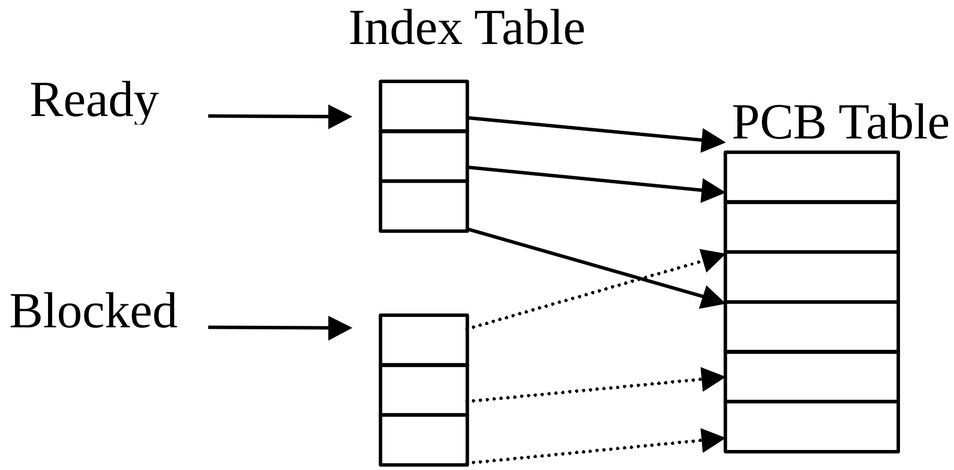
Implementation of Processes (3)

- PCB and State Queues
  - PCBs are data structures, dynamically allocated in OS memory.
  - When a process is created, a PCB is allocated to it, initialized, and placed on the correct queue.
  - As the process computes, its PCB moves from queue to queue.
  - When the process is terminated, its PCB is deallocated.

Implementation of Processes (4)

- Process Table or PCB Table
  - Os maintain a process table, each entry is PCB.
  - The size of PCB table determine the concurrency degree of system.
  - Two organization form
    - Link
    - Index



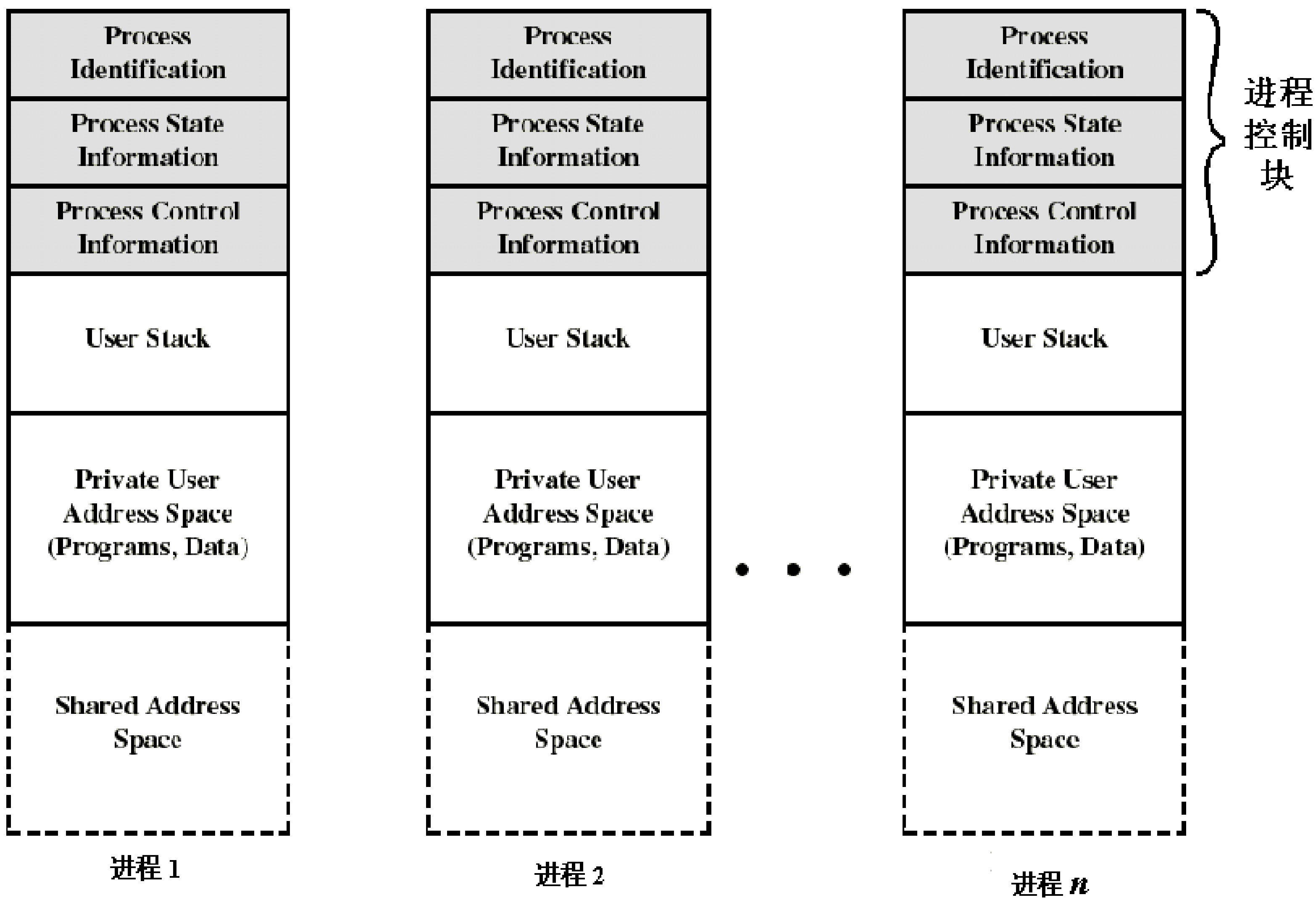


Implementation of Processes (5)

- What is In a Process? (Process Image)
  - User Code
  - User Data
  - User Stack
    - Used for procedure call and parameter passing.
  - PCB (Metadata)

Implementation of Processes (6)

在虚存中的进程映象

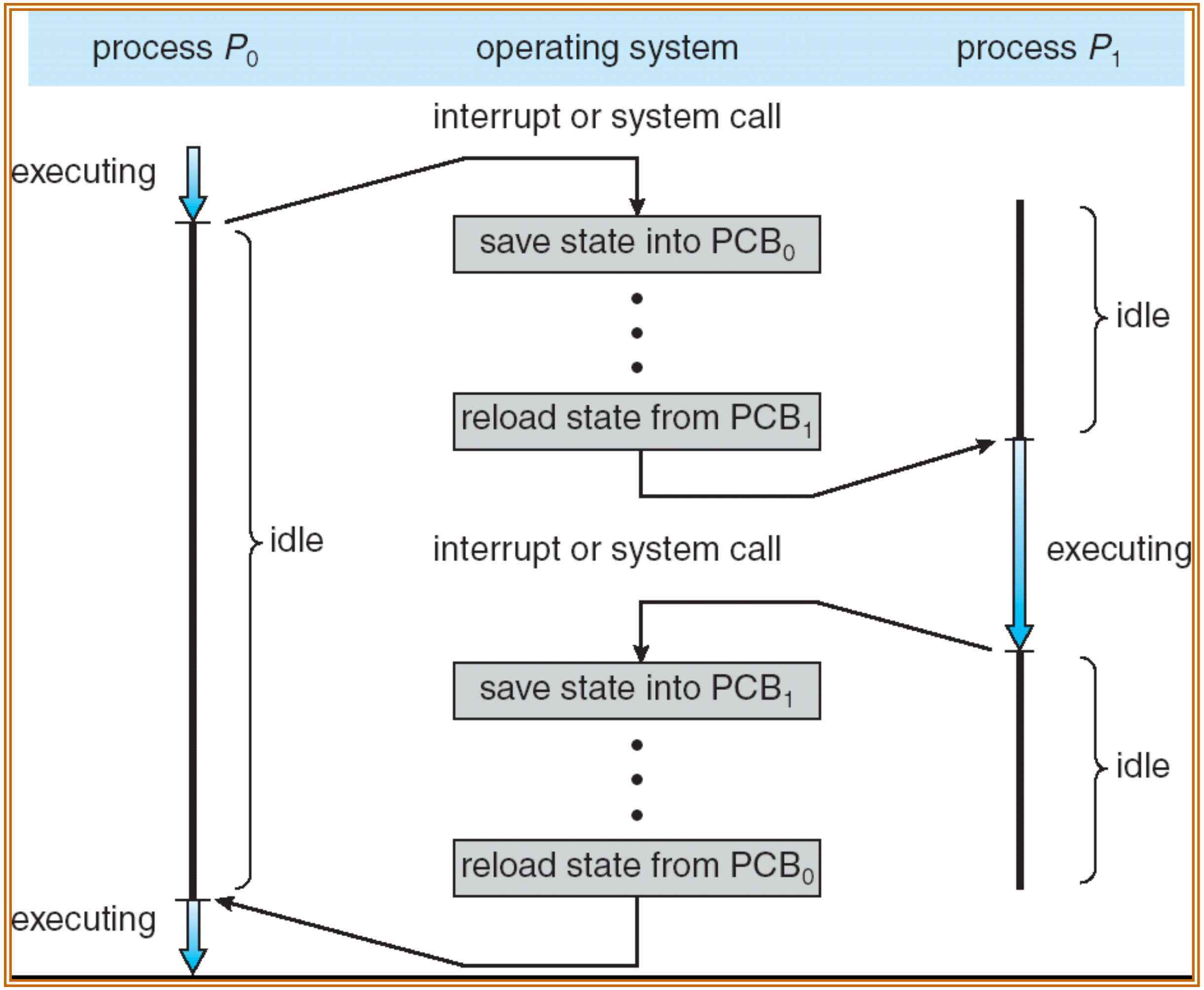


Context Switch (1)

- Switch CPU from one process to another.
- Performed by scheduler (chapter 2.4).
- It includes:
  - Save PCB state of the old process;
  - Load PCB state of the new process;
  - Flush memory cache;
  - Change memory mapping (TLB);
- Context switch is expensive(1-1000 microseconds)
  - No useful work is done (pure overhead).
  - Can become a bottleneck.
  - Real life analogy?
- Context switch overhead in Linux 2.4.21
  - About 5.4 usec on a 2.4 GHz Pentium 4
  - This is equivalent to about 13,200 CPU cycles!
    - *Not quite that many instructions since CPI > 1*
- Need hardware support.

Context Switch (2)

- CPU Switch From Process to Process



Process Summary (1)

- What Is A Process?
  - It’s one executing instance of a “program”.
  - It’s separate from other instances.
  - It can start (“launch”) other processes.
- What’s in a process?
  - Code (text)
  - Data
  - Stack
  - Process control block

Process Summary (2)

- Process vs. Program
- Process Creation
- Process Termination
- Process States
  - Running, Ready, Blocked
  - State Transition
- Context Switch

Exercise (1)

- 1. A process is a (   ).
  - A. Operating system itself.
  - B. A complete software package
  - C. <u>Program in execution</u>
  - D. Interrupt handler

Exercise (2)

- 2. The state of a process is stored in its (   ).
  - A. registers
  - <u>B. PCB</u>
  - C. source code
  - D. memory

Exercise (3)

- 3. In operating system, each process has its own（　　　）.
  - A. address space and global variables
  - B. open files
  - C. pending alarms, signals and signal handlers
  - D. all of the above

Exercise (4)

- 4. Ready state of a process means （　　　）.
  - A. when process is scheduled to run after some execution
  - B. when process is unable to run until some task has been completed
  - C. when process is using the CPU
  - D. when process is stopped

Exercise (5)

- 5. The switching of CPU between different processes is called (   ).
  - A. Swapping
  - B. Organizing
  - C. <u>Context Switching</u>
  - D. Multiple Switching

- 2.1 Processes
- 2.2 Threads

Concurrency in Processes (1)

- Many programs want to do many things “at once”
  - Microsoft Word
    - Reading and interpreting keystrokes
    - Formatting line and page breaks
    - Displaying what you typed
    - Spell checking
    - Hyphenation
    - ….
  - Web Browser
    - Download web pages, read cache files, accept user input, ...
  - Web Server
    - Handle incoming connections from multiple clients at once.
  - Scientific Programs
    - Process different parts of a data set on different CPUs.

Concurrency in Processes (2)

- In each case, would like to *share memory* across these activities.
  - Microsoft Word
    - Share same file.
  - Web Browser
    - Share buffer for HTML page and inlined images.
  - Web Server
    - Share memory cache of recently-accessed pages.
  - Scientific Programs
    - Share memory of data set being processes.

Why Need Threads? (1)

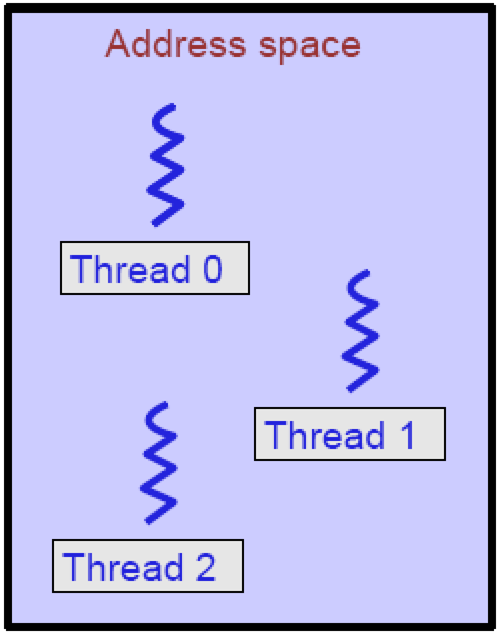
- Concurrency in Processes
  - Many activities are going on at once.
  - Some of these activities may block from time to time.
- Processes are not very efficient.
  - Each process has its own PCB and OS resources.
  - Typically high overhead for each process: e.g., 1.7 KB per task_struct on Linux!
  - Creating a new process is often very expensive.

Why Need Thread? (2)

- Processes don't (directly) share memory.
  - Each process has its own address space.
  - Parallel and concurrent programs often want to directly manipulate the same memory.
    - *E.g., When processing elements of a large array in parallel.*
- Performance Consideration
  - Overlap substantial computing and I/O activities.

Threads (1)

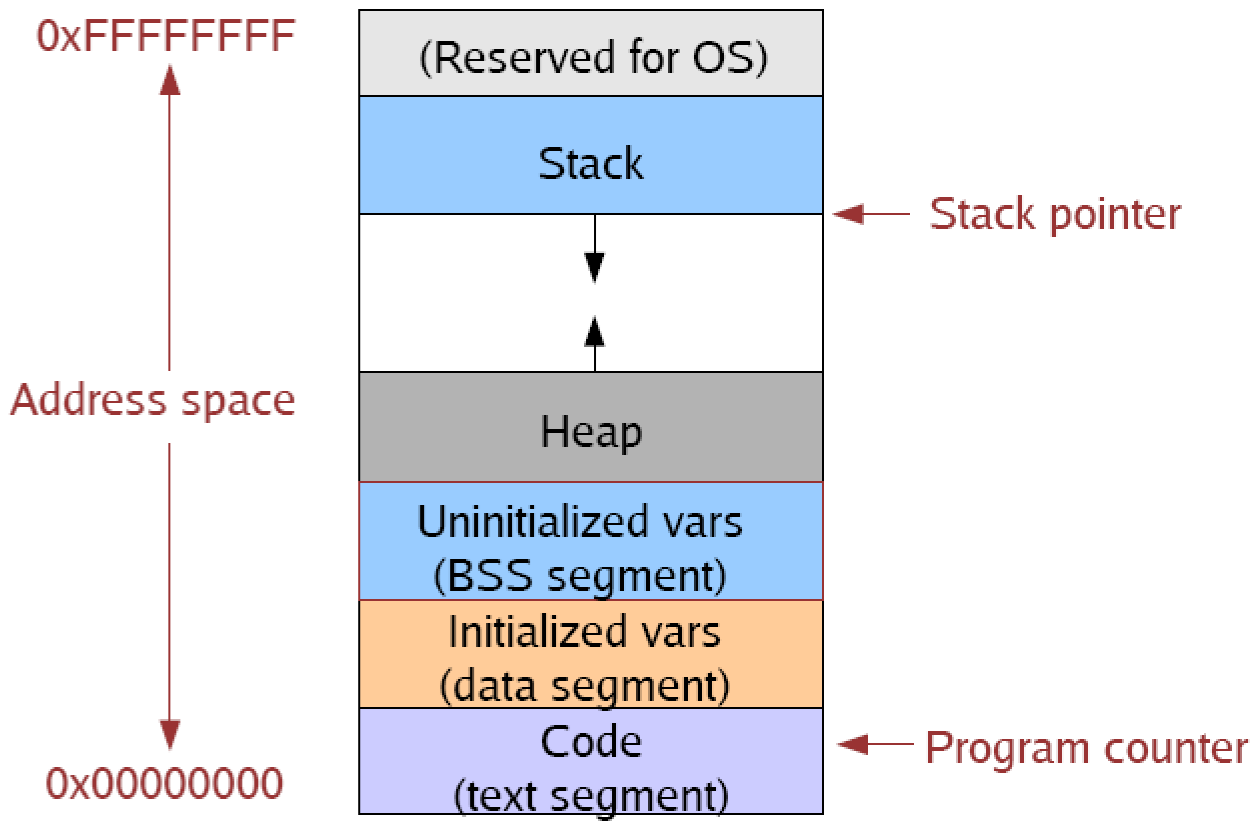
- What is a thread?
  - A sequential execution stream within a process.
- Threads separate the notion of execution from the Process abstraction.



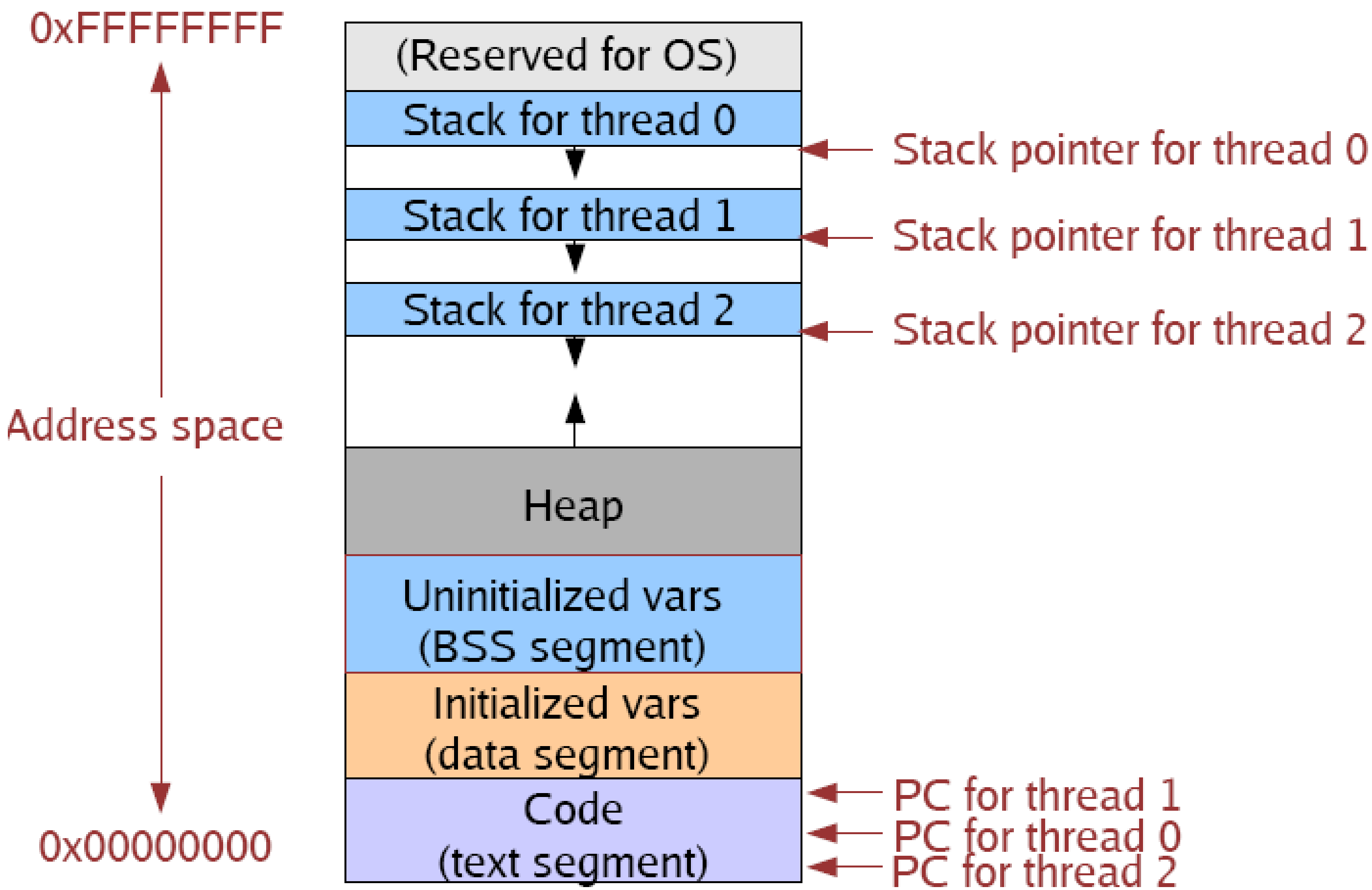
Threads (2)

- Each process has one or more threads "within" it.
  - All threads within a process share the same address space and OS resources.
    - *Threads share memory, so they can communicate directly!*
  - Each thread has its own stack, CPU registers, etc.

(Old) Process Address Space



(New) Process Address Space with Threads



The Classical Thread Model (1)

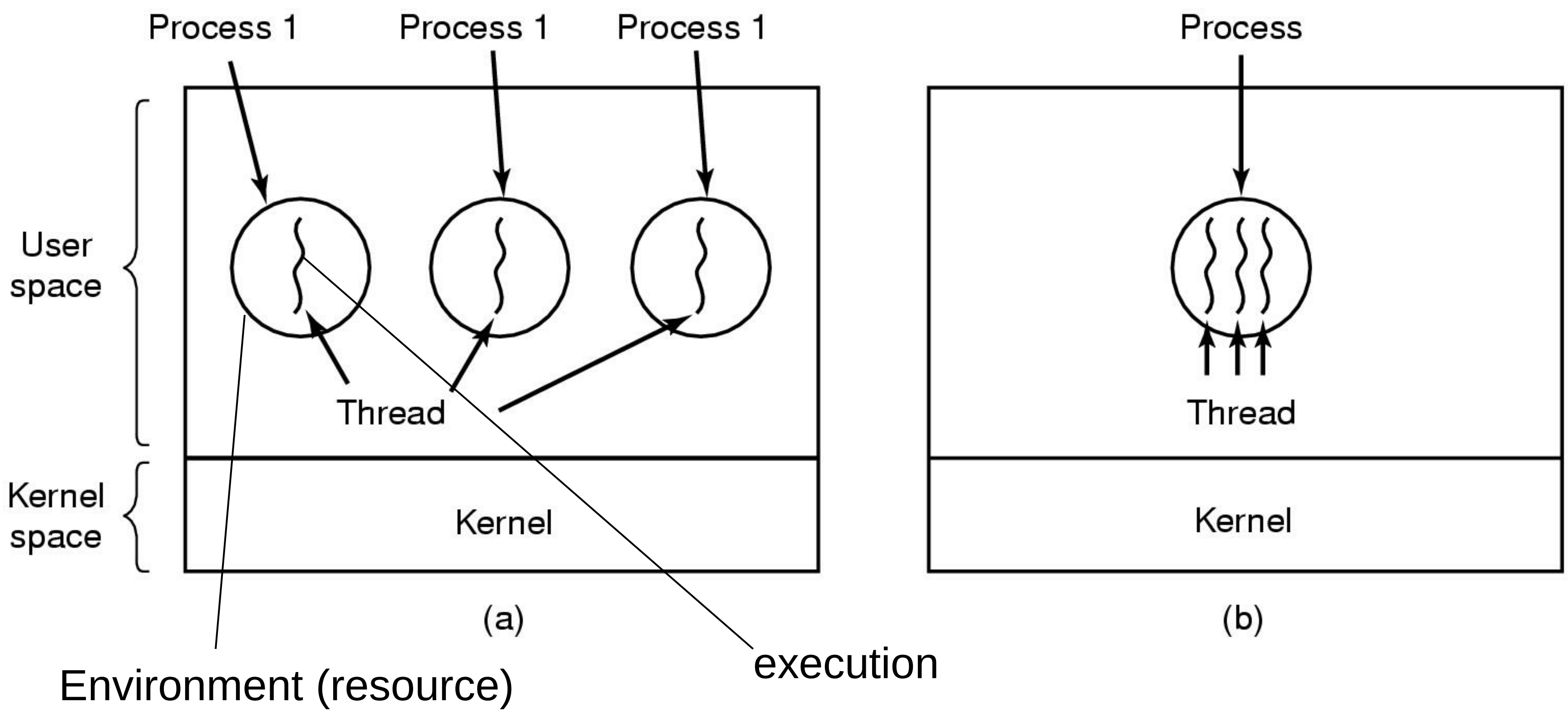


Figure 2-11

(a) Three processes each with one thread

(b) One process with three threads Multi-thread

The Classical Thread Model (2)

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Threads in the same process share resources.
- Each thread execute separately.

The Classical Thread Model (3)

- Why each thread has its own stack?

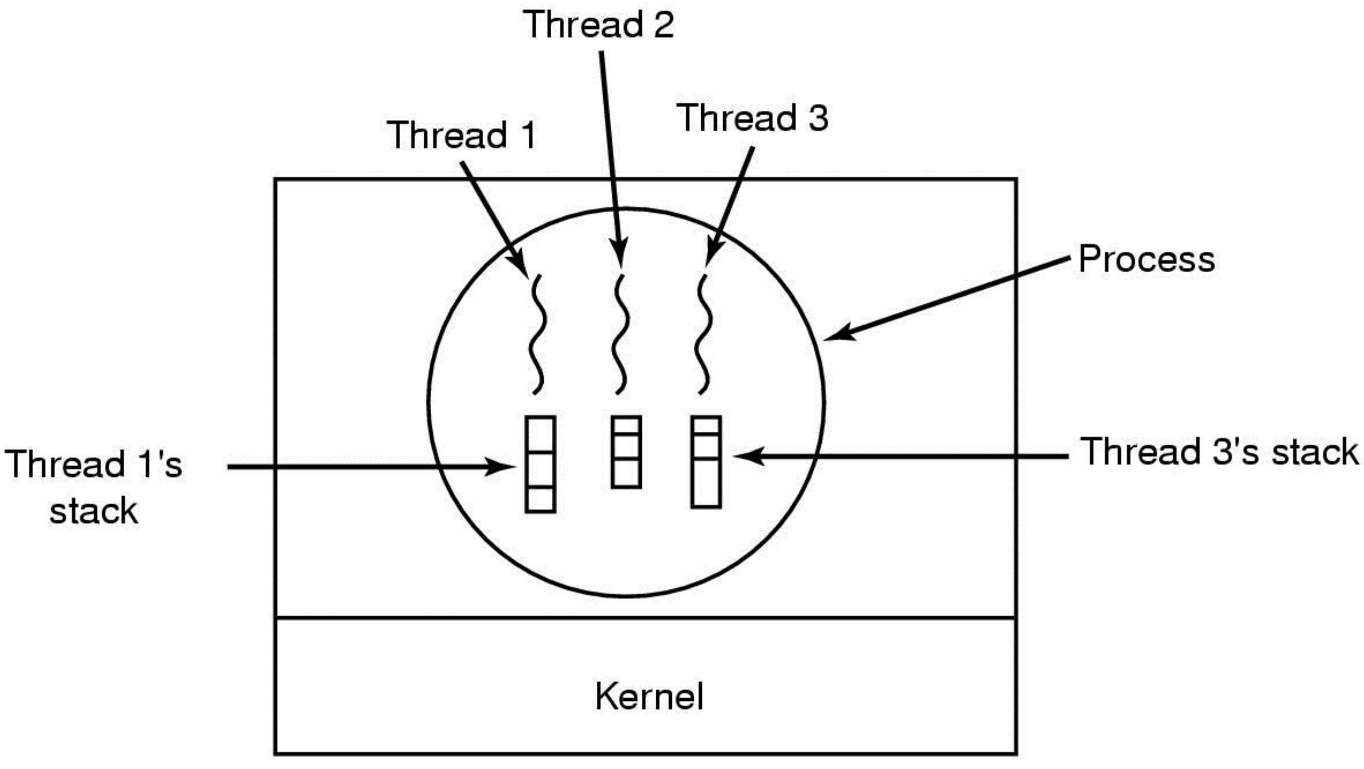
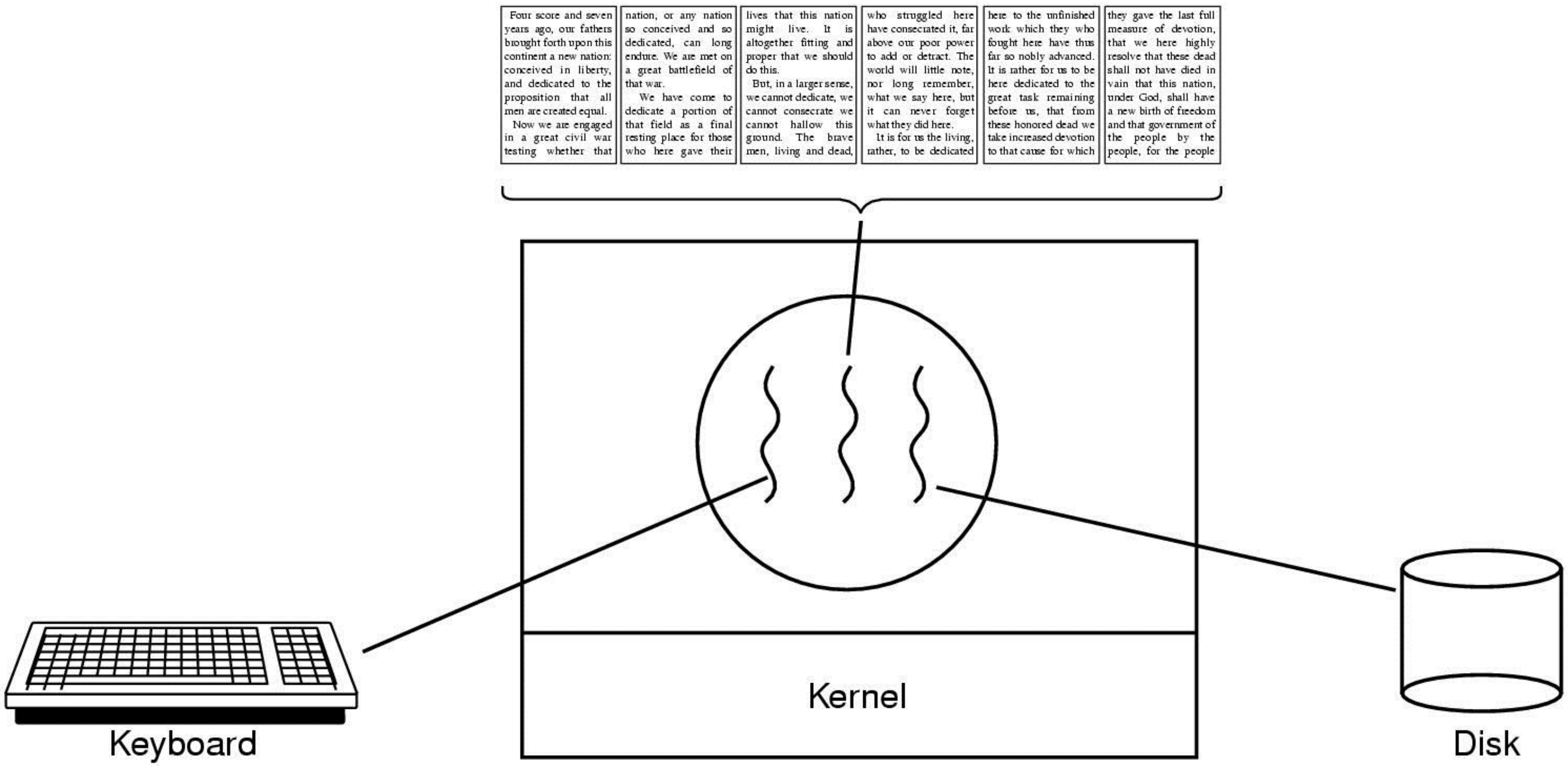


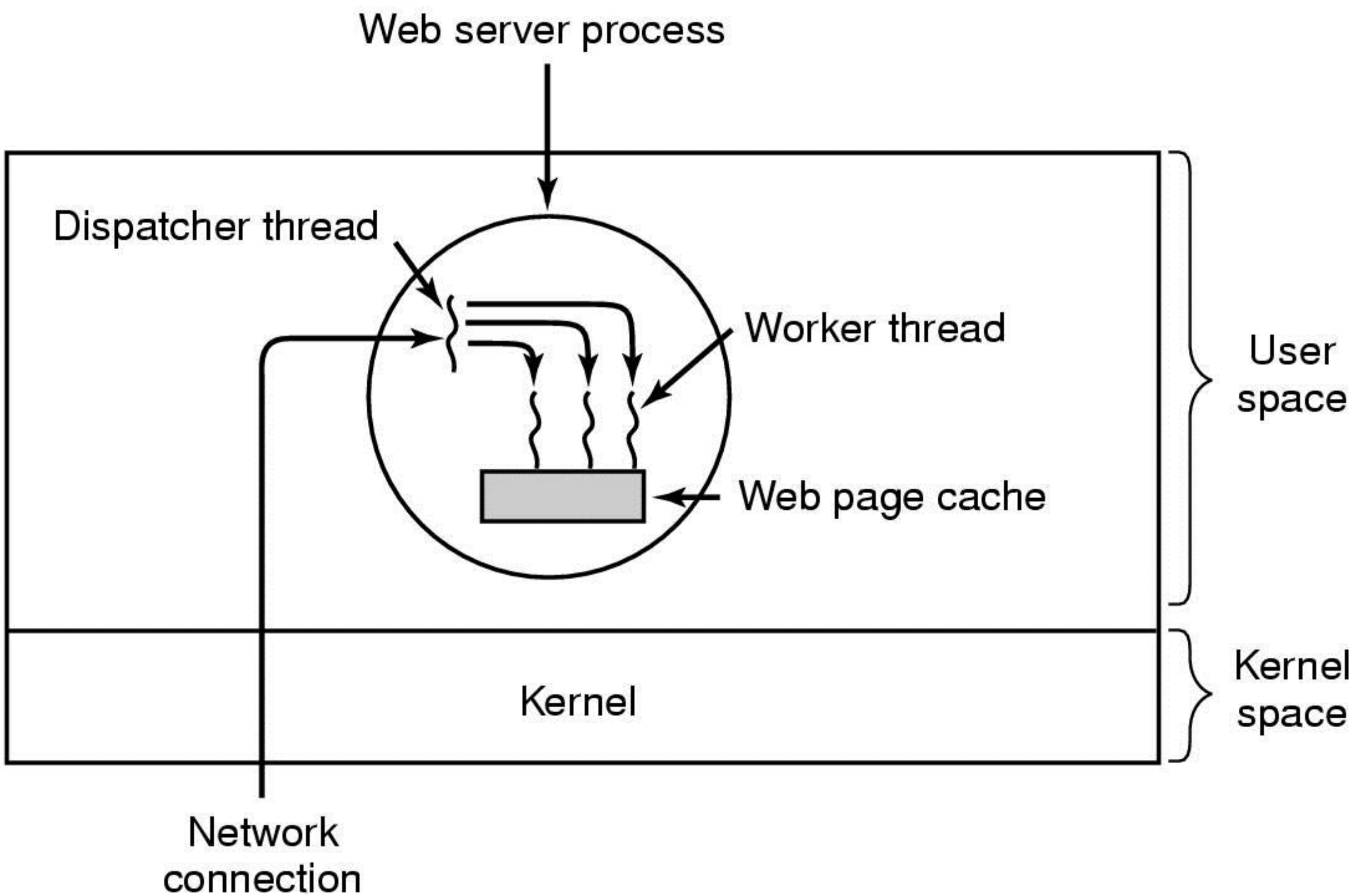
Figure 2-13. Each thread has its own stack

Thread Usage: Word Processor



- Three threads: interactive thread, reformatting thread, disk backup thread
- What if it is single-threaded?

Thread Usage: Multithread Web Server (1)



Thread Usage: Multithread Web Server (2)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

blocking system call

Alternatives: Single Threaded Web Server Implementation (1)

- Sequential Processing of Requests
  - Gets request, processes it, gets next.
  - CPU idle while data is retrieved from disk.
  - Poor performance
- Finite State Machine (IRP/event-driven program model)
  - Use non-blocking system calls (read)
  - Record state of current request
  - Event: Get next request
  - Event: On reply from disk (signal/interrupt) process data read
  - Acceptable performance
  - Complicated to develop, debug …

Alternatives: Single Threaded Web Server Implementation (2)

- FSM Algorithm

```
Loop //event loop
    get_nxt_event
    if event is a web-request
        ...
        if not in web-cache
            update req-tbl
            non-blking-read()
    else if event is reply from disk
            update req-tbl
            send reply to client
//end loop
```

Non-blocking call, i.e., read and event loop
Execute concurrently.

Alternatives: Single Threaded Web Server Implementation (3)

- Tradeoffs

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-10. Three ways to construct a server

Blocking Vs. Non-blocking System Calls

- Blocking system call
  - Usually I/O related: read(), fread(), getc(), write()
  - Doesn’t return until the call completes.
  - The process/thread is switched to blocked state.
  - When the I/O completes, the process/thread becomes ready.
  - Simple
  - Real life example: attending a lecture
- Using non-blocking system call for I/O
  - Asynchronous I/O
  - Complicated
  - The call returns once the I/O is initiated, and the caller continue.
  - Once the I/O completes, an interrupt is delivered to the caller.
  - Real life example: apply for job

Benefits of Threads

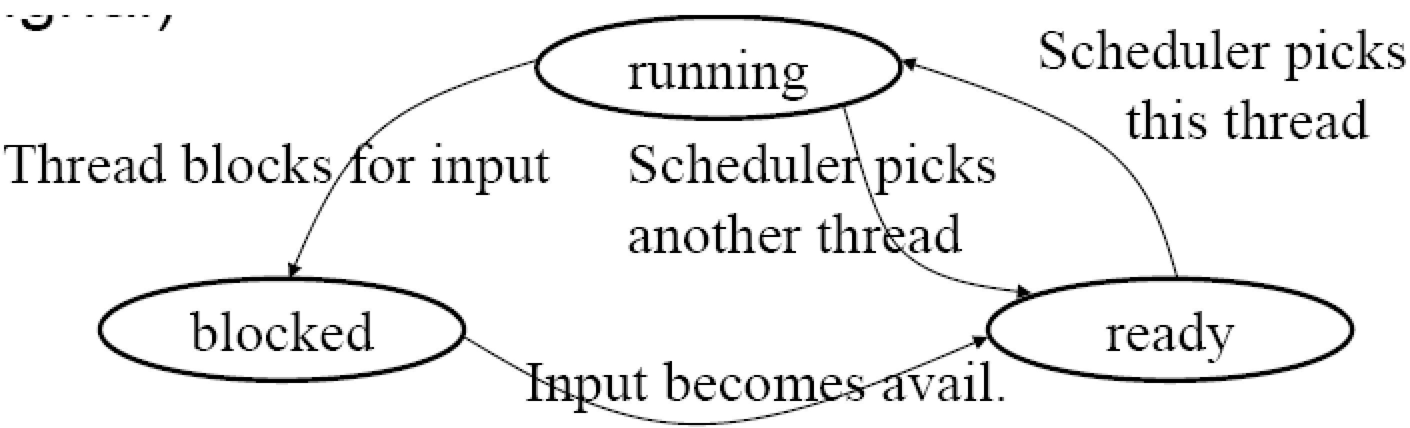
- Responsiveness
  - Multithreading an interactive application allows a program to continue running even if part of it is blocked or performing a lengthy operation.
- Resource Sharing
  - Sharing resources may result in efficient communication and high degree of cooperation.
- Economy
  - Thread is more light weight than process.
- Scalability
  - Better utilization of multiprocessor architectures.

Common Thread Interface

- thread_create(…): creates a thread
- thread_wait(…): waits for a specific thread to exit
- thread_exit(…): terminates the calling thread
- thread_join(…): blocks the calling thread until a (specific) thread has exited
- thread_yield(…): calling thread passes control on voluntarily to another thread

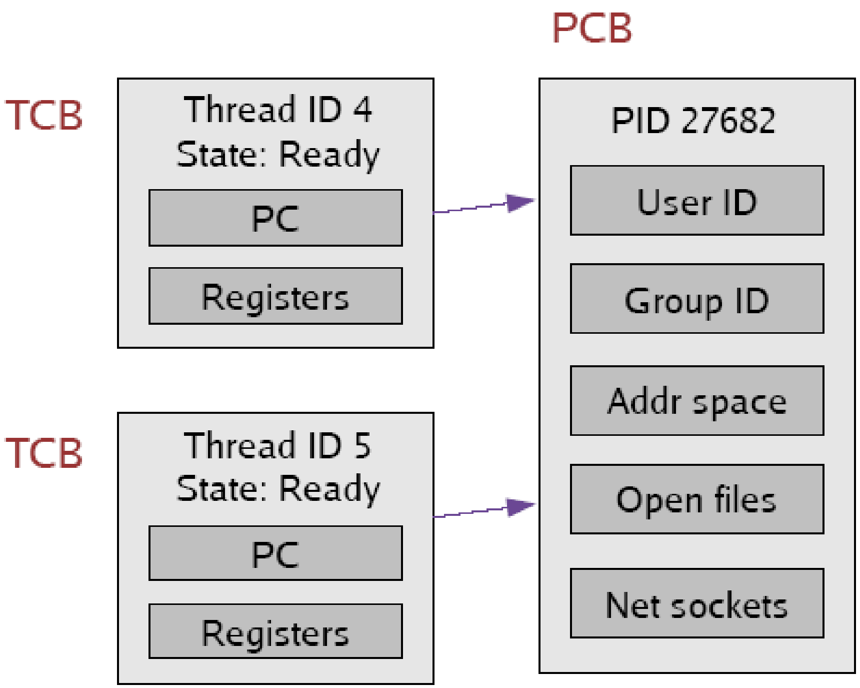
Thread States

- Three states (implementation dependent):
  - Running: currently active and using CPU.
  - Ready: runnable and waiting to be scheduled.
  - Blocked: waiting for an event to occur (I/O, signal).



Thread Control Block (1)

- Given what we know about processes, implementing threads is“easy”.
- Idea: Break the PCB into two pieces
  - Thread-specific stuff: Processor state
  - Process-specific stuff: Address space and OS resources (open files, etc.)

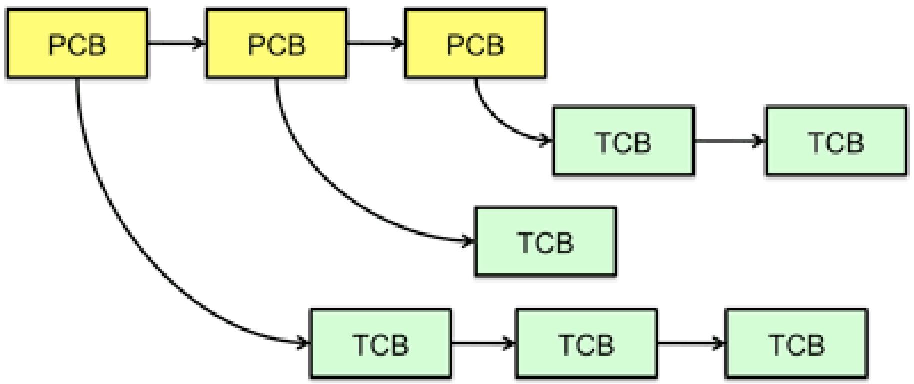


Thread Control Block (2)

- TCB contains info on a single thread
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB)
  - Etc (add stuff as you find a need)
- PCB contains information on the containing process
  - Address space and OS resources ... but NO execution state!
- TCB's are smaller and cheaper than processes
  - Linux TCB (thread_struct) has 24 fields
  - Linux PCB (task_struct) has 106 fields

Thread Control Block (3)

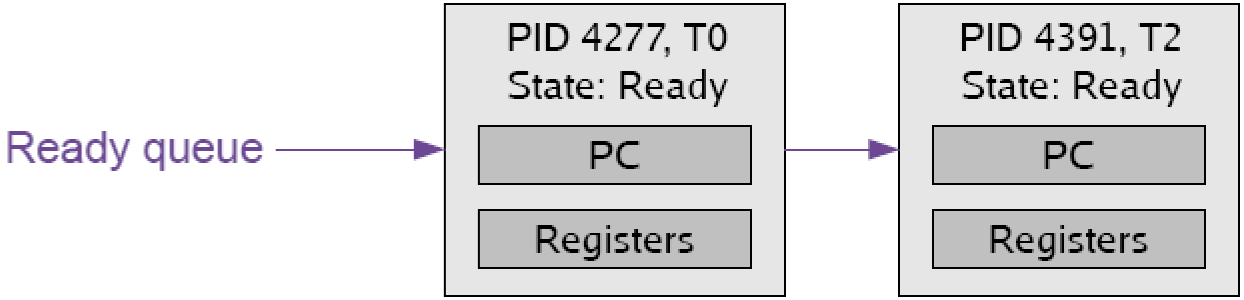
- PCB points to multiple TCBs:



- Switching threads within a block is a simple thread switch.
- Switching threads across blocks requires changes to memory and I/O address tables.

Context Switching (1)

- TCB is now the unit of a context switch.
  - Ready queue, wait queues, etc. now contain pointers to TCB's.
  - Context switch causes CPU state to be copied to/from the TCB.



Context Switching (2)

- Context switch between two threads in the *same* process:
  - No need to change address space.
- Context switch between two threads in *different* processes:
  - Must change address space, sometimes invalidating cache.
  - This will become relevant when we talk about virtual memory.

Group Discussions (3 minutes)

- Similarity between processes and threads

- Difference between processes and threads

- Real life examples?

Similarities between Process & Thread

- Like processes, threads share CPU and only one thread is running at a time.
- Like processes, threads within a process execute sequentially.
- Like processes, threads can create children.
- Like a traditional process, a thread can be in any of several states: running, ready, or blocked.
- Like processes, threads have program counter, stack, registers and state.

Dissimilarities between Process & Thread

- Unlike processes, threads are not independent of one another.
- Threads within a process share an address space.
- Unlike processes, threads are designed to assist one another. Note that processes might or might not assist one another because processes may be originated from different users.

Process vs. Thread

- Process
  - **Unit of resource ownership** with respect to the execution of a single program.
  - Can encompass **more than one thread of execution.**
  - Processes are largely independent.
- Thread
  - **Unit of execution**
  - Belongs to a process.
  - Threads are part of the same “job” and are actively and closely cooperating.

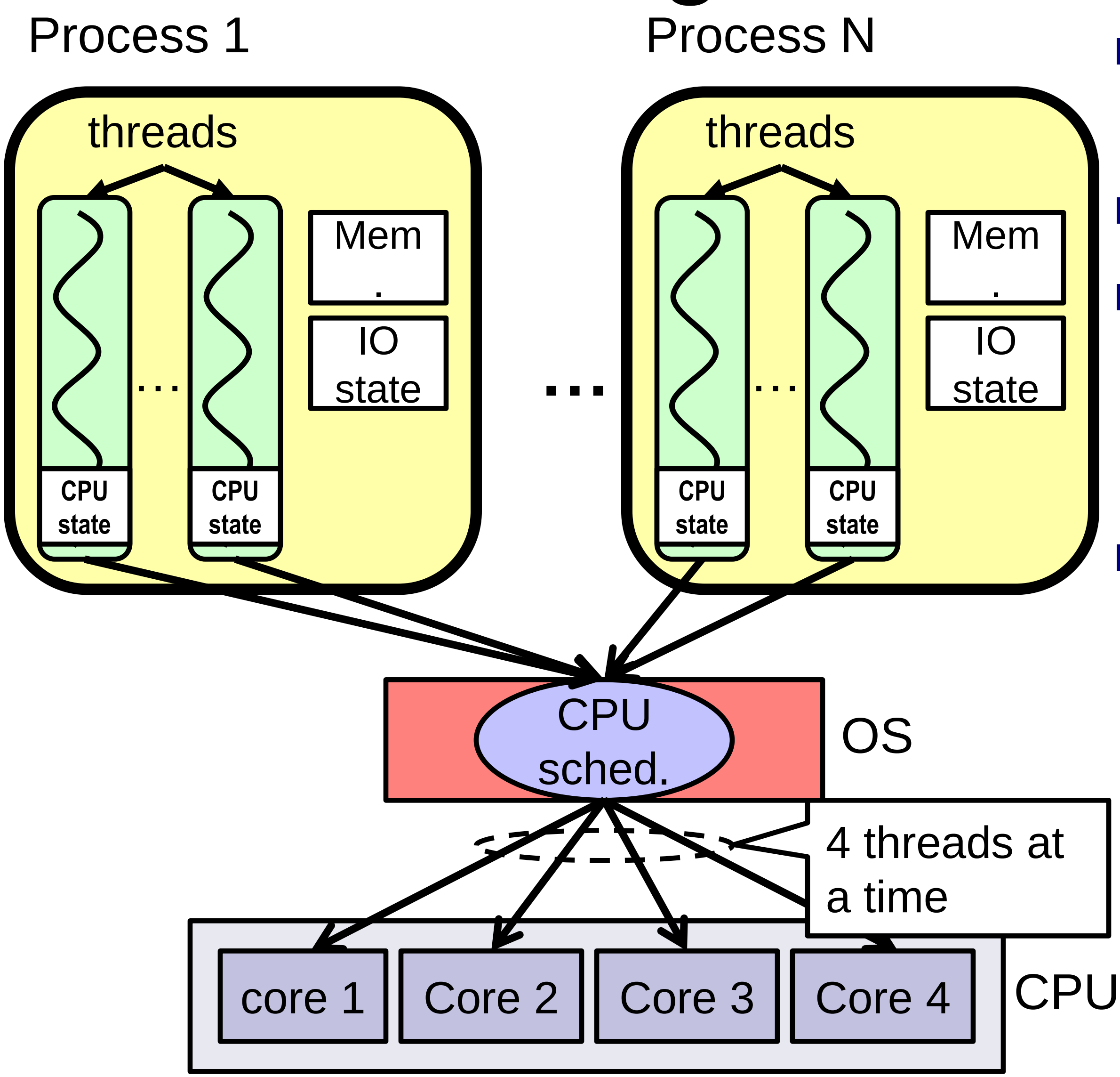
Real Life Example

- Process
  - OS Project
  - Different from DS's Project.
- Thread
  - OS Project1, Project2
  - Each is different.
  - Share
    - Nachos
    - Textbook
    - Personnel (TAs, instructors)
  - Affect each other.

Multithreading and Multicore (1)

- Multithreading
  - Multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.

Multithreading and Multicore (2)



- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: No
- Sharing overhead: **low** (thread switch overhead low)

POSIX Threads (1)

- IEEE standard 1003.1c
  - Defines a standard operating system interface and environment to support applications portability at the source code level.
- Pthread
  - API for thread creation and synchronization.
  - Over 60 function calls.
  - API specifies behavior of the thread library, implementation is up to development of the library.
  - Not concerned with the details of the implementation – can be implemented at the OS level or at the application level.
  - Common in UNIX operating systems.

POSIX Threads (2)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

POSIX Threads (3)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Figure 2-15. An example program using threads.

Where do Threads Come From?

- A few choices:
  - A user-mode library
  - The operating system
  - Some combination of the two…

User-level Threads (1)

- To make threads cheap and fast, they need to be implemented at user level.
  - User-level threads are managed entirely by the run-time system (thread library).
- User-level threads are small and fast.
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB).
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call.
    - No kernel involvement.
  - User-level thread operations 100x faster than kernel threads.

User-level Threads (2)

- Implementing Threads in User Space (old Linux)
  - Implemented as a library inside a process.
  - All operations (creation, destruction, yield) are normal procedure calls.

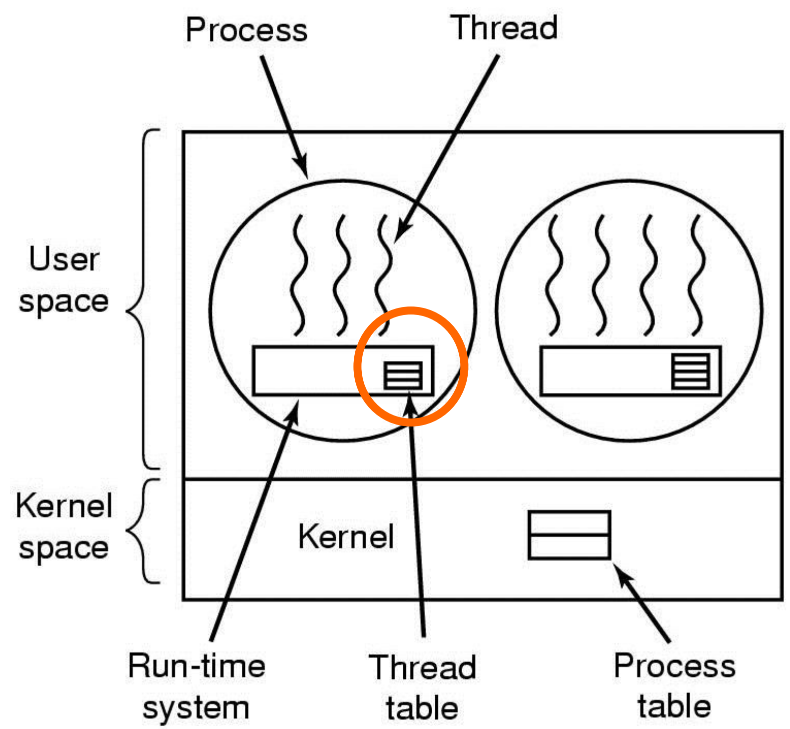


Figure 2-16. (a) A user-level threads package

User-level Threads (3)

- Advantages of User-level Threads
  - ULTs can run on any OS.
    - Only needs a thread library.
  - ULTs are fast to create and manage.
  - Thread context switch is much faster than process context switch.
    - No kernel mode switch required.
    - Only registers are saved/loaded.
    - Thread may voluntarily yield (local call).

User-level Threads (4)

- Advantages of User-level Threads (ctd.)
  - Scheduling can be application specific in the user level thread.
    - May support per-process customized scheduling algorithms.
  - Better scalability.
    - i.e., more threads, since no kernel access and data structures are required.

User-level Threads (5)

- Issues of User-level Threads
  - Most system calls are blocking and the kernel blocks processes. If one user level thread perform blocking operation then entire process will be blocked.
  - Solutions
    - 1. Could change system calls to be all non-blocking.
      - Requires major rewrite of OS.
    - 2. Tell in advance if a call will block.
      - Requires rewriting parts of the system call library.

User-level Threads (6)

- Issues of User-level Threads (ctd.)
  - Threads may monopolize CPU (no clocks in process).
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors .

Kernel-level Threads (1)

- We have taken the execution aspect of a process and separated it out into threads.
  - To make concurrency cheaper.
- As such, the OS now manages threads *and* processes.
  - All thread operations are implemented in the kernel.
  - The OS schedules all of the threads in the system.
- OS-managed threads are called kernel-level threads or lightweight processes.
- Most modern OSes support kernel-level threads.
  - Windows XP/2000, Solaris
  - Linux , Tru64 UNIX
  - Mac OS X

Kernel-level Threads (2)

- Implementing Threads in the Kernel
  - Threads implemented inside the OS
    - Thread operations (creation, deletion, yield) are system calls.
    - Scheduling handled by the OS Scheduler.

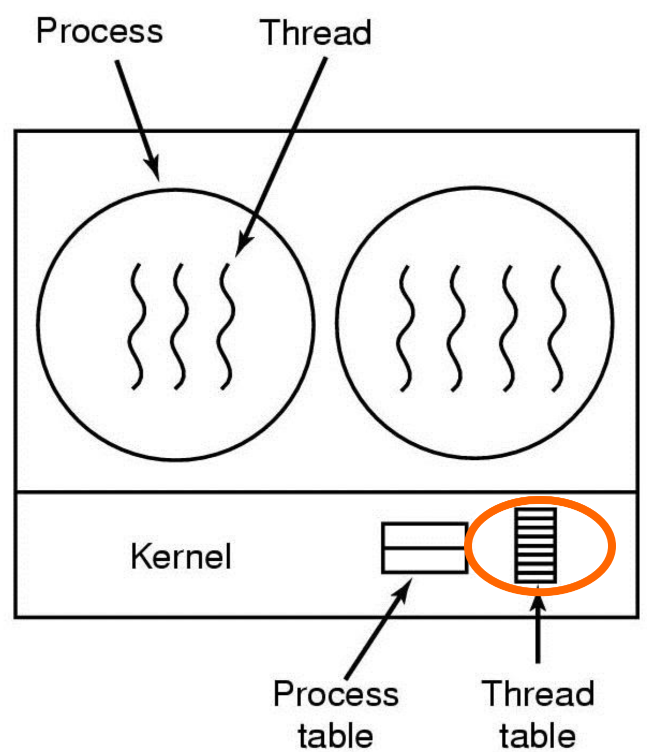


Figure 2-16. (b) A threads package managed by the kernel

Kernel-level Threads (3)

- Advantages of Kernel-level Threads
  - If one thread in a process is blocked, the kernel can schedule another thread of the same process.
  - Kernel routines themselves can be multithreaded.
  - Kernel can schedule multiple threads on different CPUs on SMP multiprocessor.

Kernel-level Threads (4)

- Disadvantages of Kernel-level Threads
  - Slower to schedule, create, delete than user-level threads.
  - Transfer of control from one thread to another within the same process requires a mode switch to the kernel.
  - OS must scale well with increasing number of threads.

Differences between ULTs & KLTs

User-level Threads	Kernel-level Threads
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
Operating system supports creation of Kernel threads.	Kernel-level thread is specific to the operating system.
Can not be scheduled by OS.	Can be scheduled by OS.
Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Hybrid Implementations (1)

- Some operating system provide a combined user level thread and kernel level thread facility.
- ***Solaris*** is a good example of this combined approach.
- Hybrid of kernel and user-level threads: m-to-n thread mapping
  - Application creates m threads.
  - OS provides pool of n kernel threads.
  - Few user-level threads mapped to each kernel-level thread.

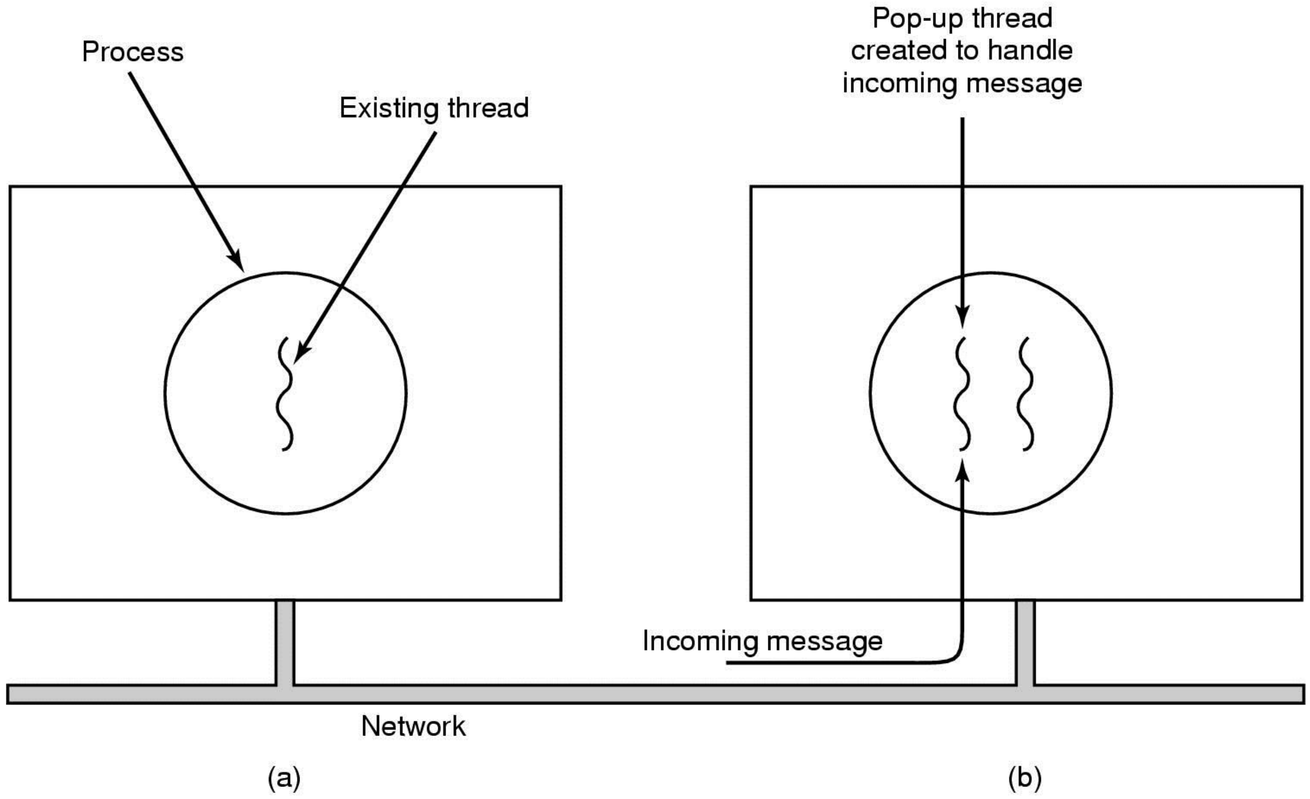
Hybrid Implementations (2)

- Advantages
  - Can get best of user-level and kernel-level implementations.
  - Works well given many short-lived user threads mapped to constant-size pool.
- Disadvantages
  - Complicated…
  - How to select mappings?
  - How to determine the best number of kernel threads?
    - User specified
    - OS dynamically adjusts number depending on system load.

Pop-Up Threads (1)

- Pop-Up Threads
  - Created on “as needed” basis.
- Advantages of Pop-Up Threads
  - Cheaper to start new thread than to restore existing one.
  - No threads block waiting for new work; no context has to be saved, or restored.

Pop-Up Threads (2)



- Creation of a new thread when message arrives
  - (a) before message arrives
  - (b) after message arrives

Thread Summary

- Why need threads?
- What is thread?
- Differences between process and thread.
- Thread Implementations and their tradeoffs
  - User-level Threads
  - Kernel-level Threads
  - Differences between ULTs & KLTs
  - Hybrid Implementation
  - Pop-up Threads

Exercise (1)

- 1. Which one of the following is not shared by threads? (   ).
  - A. program counter
  - B. stack
  - C. both A and B
  - D. none of the above

Exercise (2)

- 2. The time required to create a new thread in an existing process is: (   ).
  - A. greater than the time required to create a new process
  - B. <u>less than the time required to create a new process</u>
  - C. equal to the time required to create a new process
  - D. none of the above

Exercise (3)

- 3. Because the kernel thread management is done by the Operating System itself: (   ).
  - A. kernel threads are faster to create than user threads
  - B. kernel threads are slower to create than user threads
  - C. kernel threads are easier to manage as well as
  - D. none of the above

Exercise (4)

- 4. Which of the following is FALSE ? (   ).
  - A. Context switch time is longer for kernel level threads than for user level threads.
  - B. User level threads do not need any hardware support.
  - C. Related kernel level threads can be scheduled on different processors in a multiprocessor.
  - D. Blocking one kernel level thread blocks all other related threads.

Homework

- P174 1,12,16,18