Operating Systems

CSE 231

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 2: Aug 31 – Sep 3

What is a program (binary) – under the hood

- Program compilation:
  - Preprocessing: Text replacement.
  - Lexical Analysis: Identification of the lexemes/tokens/keywords.
  - Parsing/Syntax Analysis: Checking if the program is correctly generated as per the grammar (production rules)
  - Semantic Analysis: Associating further actions based on the matched production rules.
  - IR generation: Generate IR (machine like code) for the actions.
  - IR □ Assembly language
  - Assembly language □ Assember □ Unliked binary
  - Unlinked binary □ linker □ linked binary
  - Linked binary □ runtime loader □ running program

Preprocessing

- Text replacement of macros – e.g. #define, #include.
- Happens before anything else is done.
- First step of compilation.

Lexical Analysis

- Individual tokens are separated, based on a delimiters. For C language it is a blank spaces and semicolon.
- Separation of keywords, variable, values, operators etc.
- Aka tokenization.
- Checks for any missing syntax parts, e.g. missing delimiters or operators.

Syntax Analysis / Parsing

- Each statement of the language is produced through production rules.
- Checks syntax for syntactic errors. Basically – can the statement be produced by the syntax rules ?
- Each programming language has some production rules.

Preprocessing, lexical analysis, syntax analysis together is known as the **front end** of the compiler

Semantic Analysis

- What to do of each of the statements which match the production rules of the grammar.
  - Action for each production rule.
  - ``Assign'' a meaning to each statement which matches the production rule.
  - Particularly keep ``hints'' for IR generation.
  - Generally involves creation of a syntax tree which when parsed the actions has to be IR.

IR Generation

- Take ``hint'' from semantic analysis and outputs ``fake instructions'' using dummy variable and registers.
- These are used by the appropriate assembler.
- Generally the IR has ``fake instructions'' showing instruction types
  - Most CPUs have categories of instructions – load, store, move, add, jump, etc.
  - The ``fake instructions'' are used to give ``hints'' to the IR assembly language generation process.

```
unsigned  square_int(unsigned  a)
 {  return  a*a;  }
```

```
define  i32  @square_unsigned(i32  %a)
{  %1  =  mul  i32  %a,  %a  ret  i32  %1  }
```

IR Generation

```
unsigned square_int(unsigned a)
  { return a*a; }
```

```
define i32 @square_unsigned(i32 %a)
{ %1 = mul i32 %a, %a ret i32 %1 }
```

IR ? Assembly Language

- Perform any kind of code optimization.
- Take as input the IR and generate appropriate assembly instructions.

```
define i32 @square_unsigned(i32 %a)
    { %1 = mul i32 %a, %a ret i32 %1 }
```

```
square_unsigned:
    movl 4(%esp), %eax
  imull %eax, %eax
  ret
```

Assembler

- Generates the binary opcodes corresponding to the instruction mnemonics.
- Addresses, especially dynamically linked addresses are not added during assembly. Later added during run time.

Linker

- Takes the assembly output and does two things (depending upon if you are using statically linked addresses or dynamically linked addresses)
  - Static linking – Creates a binary that includes the library files. All the library functions are finally part of the binary
  - Dynamic linking – The library is mere ``addressed'' but no added to the binary. Actual addresses are assigned at runtime.

Program Memory

- A compiled (And therefore running program) has various memory blocks to it.
  - Code: Actual program instructions.
  - Stack: For local variables, Interrupt handling and returns, call and return.
  - Heap: Dynamically allocated memory (malloc() family)
  - Data: For global variables.
  - RODATA: Constants (e.g. char *p="hello world") → goes in RODATA
  - BSS: Aka Block Starting with Symbols. Used for uninitialized data (similar to Data Section)
  - By default: other than code no other memory region can be used for executing instructions from.

Program Memory

```
#include <stdio.h>

void main()
{
 int x = 0;
 char *p = "hello world!\n\r";
 printf(p);
}
```

```
    .file   "simple_prog.c"
    .section    .rodata
.LC0:
    .string "hello world!\n\r"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $0, -4(%rbp)
    movq    $.LC0, -16(%rbp)
    movq    -16(%rbp), %rax
    movq    %rax, %rdi
    movl    $0, %eax
    call    printf
    nop
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
```

How the Computer Boots Up

- Turn up the Desktop/Laptop
- POST – power on self-test to check the functioning of the hardware – single beep (in desktops) – everything OK…Multiple beeps – something wrong with hardware (RAM, Graphics Card, Etc.)
- Basic Input Output Services (BIOS) program is loaded from EEPROM.
  - Provides basic I/O handling.
  - Change settings – Disk bootup sequence, special board features and peripherals etc.
- The BIOS loads the designated boot sector(s) of the first boot drive – aka the bootloader program into the RAM.
- Jumps to the bootloader program which takes over to load the rest of the OS.

Multi-OS Bootup

- Replaces the Master Boot Record (MBR) with a bootloader – e.g. Grand Unified Bootloader (GRUB).
- Standard BIOSes can recognize 4 primary partitions of about 2 TB each.
- ONLY one of those could be chosen to be bootable at a time (boot flag).
  - Each OS with is own bootloader.
- Multi-Bootloader replaces the Master Boot Record (MBR) for loading individual bootable partition.
  - Mult-bootloaders like GRUB recognize different FileSystems formats and are able to load specific operating systems or operating system bootloaders.
  - In case of Linux, GRUB loads the kernel (OS) file.
  - In case of Windows, GRUB jumps to the first sector of the Windows partition.

Legacy vs Unified Extensive Firmware Interface (UEFI) BIOS

- Legacy BIOS – supports 4 primary bootable part partitions of size ~2TB.
  - MBR for the disk for the entire disk.
- UEFI – supports 128 partitions of sizes of about ~9ZB.
  - MBR stored in a separate dedicated FAT32 partition that can several bootloaders.

  (UEFI BIOSes are backward compatible to legacy BIOSes).

Real and Protected Memory

- System boots up with 20-bit absolute addresses. BIOS runs with absolute addresses. MS-DOS runs at absolute addresses.

- Boot-up process switches to 32/64-bit virtual addressing.

- Paging and segmentation used by most MMUs

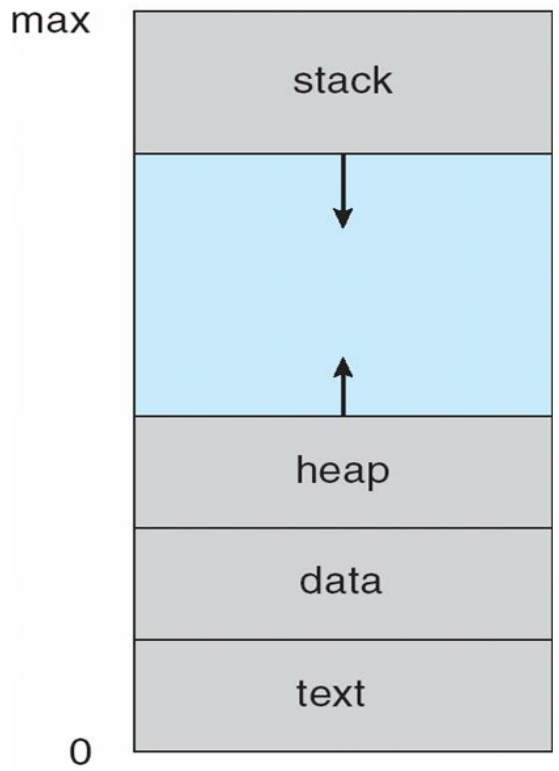
What is a Process

- For every program, the OS allocates an ``isolation'' – unique set of CODE, STACK, HEAP, DATA, RODATA and BSS for each program.
- Each program runs as if there is only itself and none other.
- The programmer need not worry about other programs in the system.
- The OS takes care of starting and stopping programs and allocation blocks of memory to each running program
- **Virtual memory:** Each process essentially uses the same virtual address space – also known as the offset address – which equals the entire address space – X86_32: 2^32 addresses, X86_64: 2^64 addresses.
- The **task scheduler/process scheduler** determines each task to start and stop and everything about the process is saved in individual **Process Control Blocks** (PCBs).

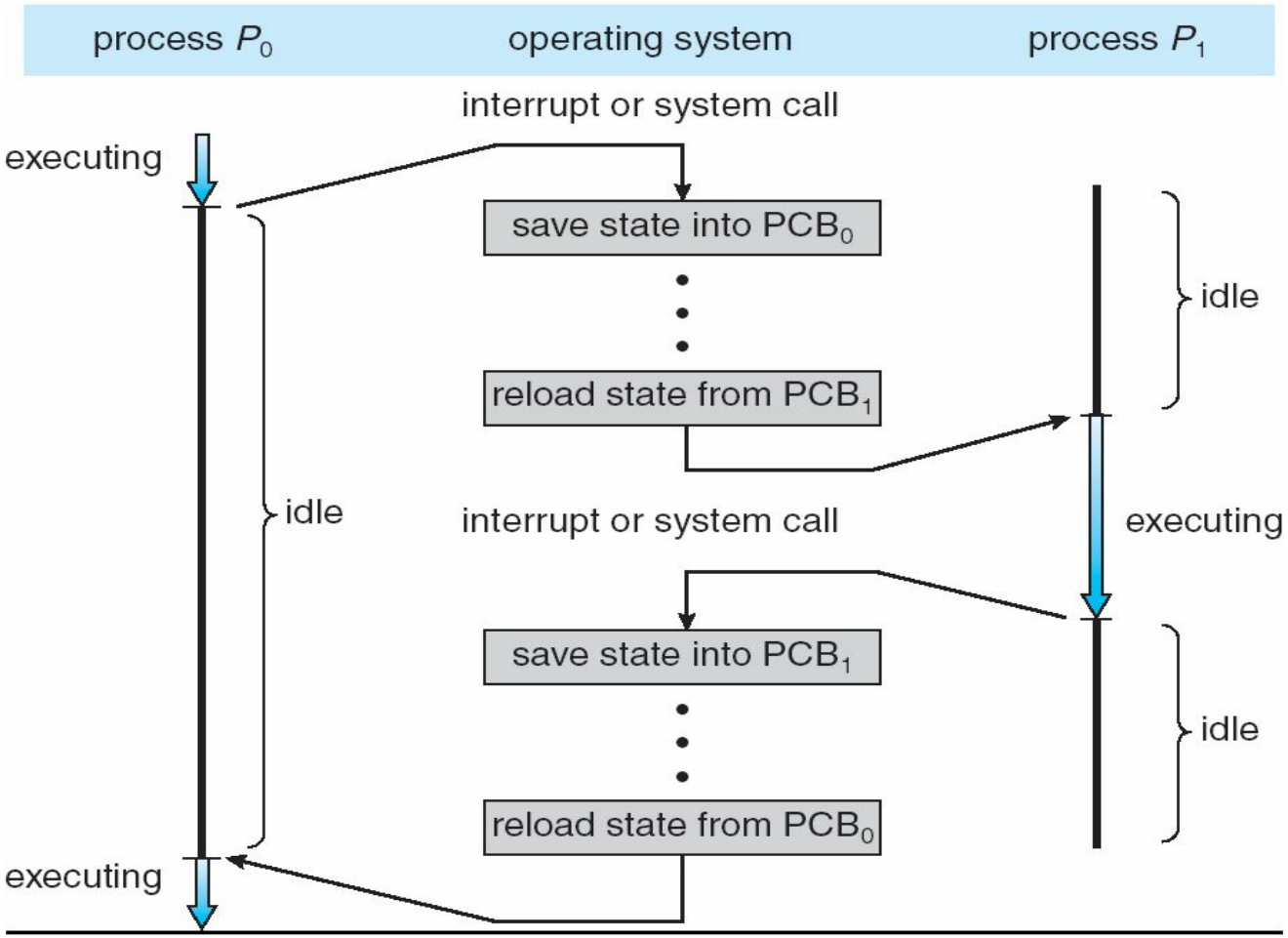
- **Each process has its own code, stack, heap, data and BSS**

Process/program memory contd.

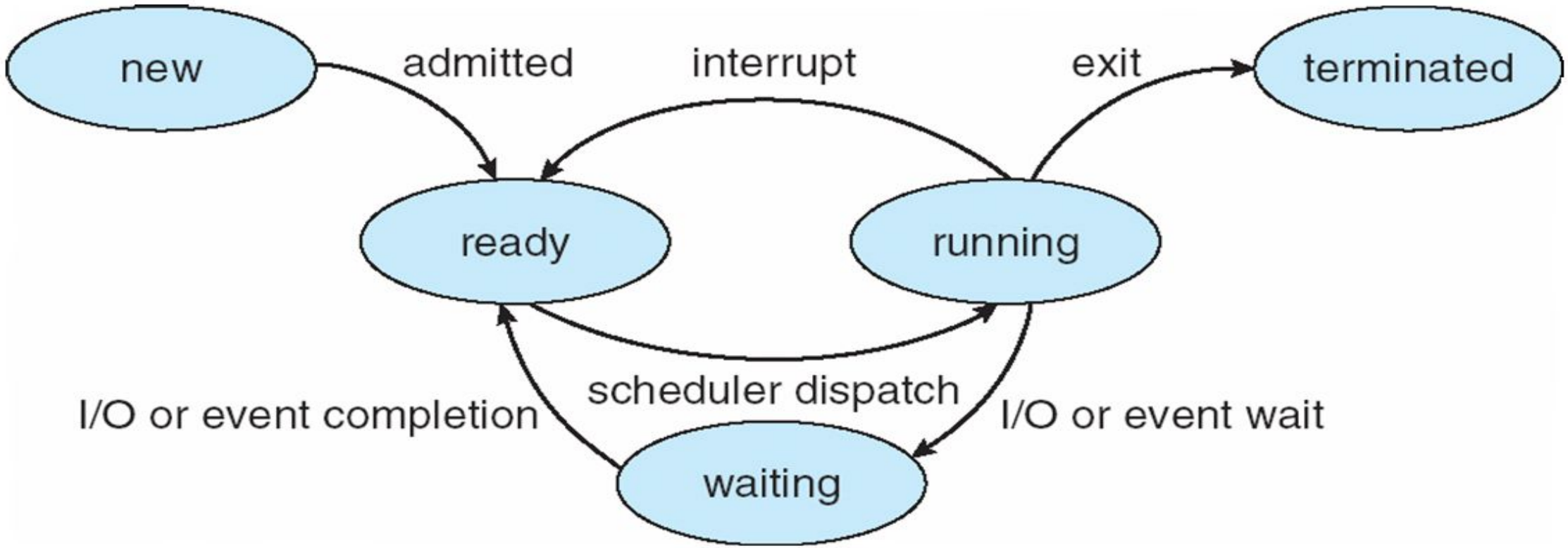
- Heap, code, data, rodata, bss – addresses increase from low to high – the registers (and subsequently the instructions) used to address it increment in the increasing order.
- Stack – address increase from higher to lower. Stack registers are decremented everytime the CPU executes a stack operation PUSH/POP/RET/CALL



Task/Process Scheduler Context Switcher



Process State

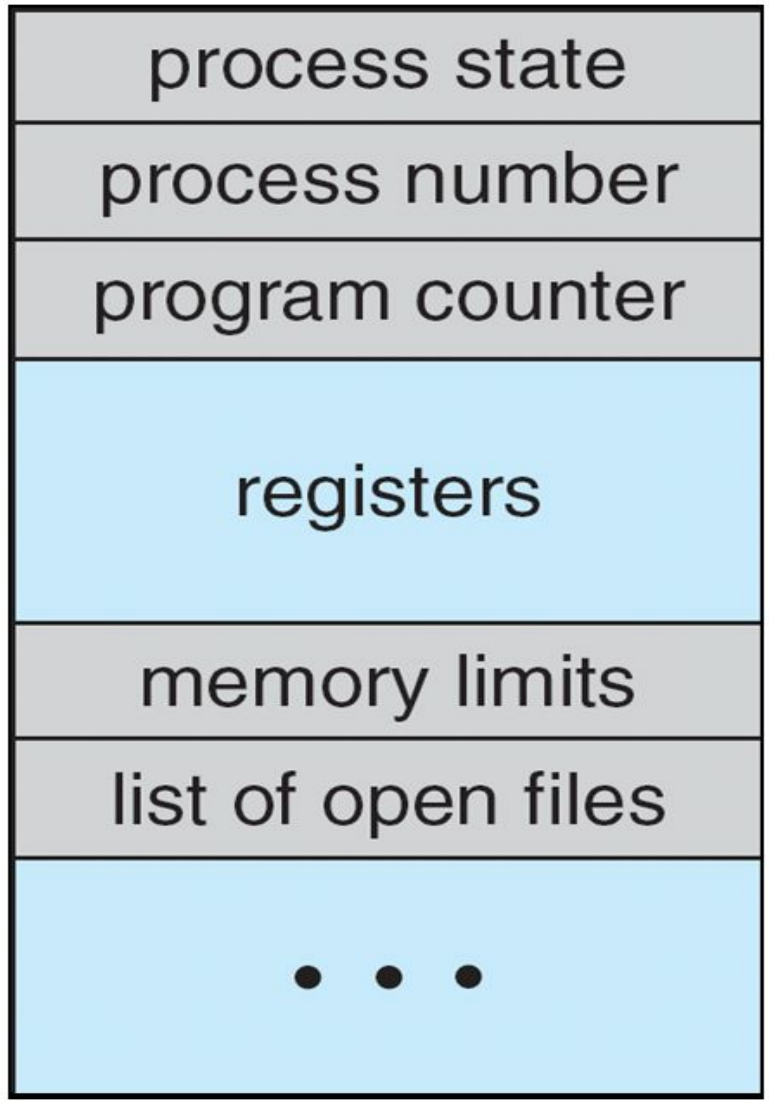


Process Control Block

Information associated with each process

(also called **task control block**)

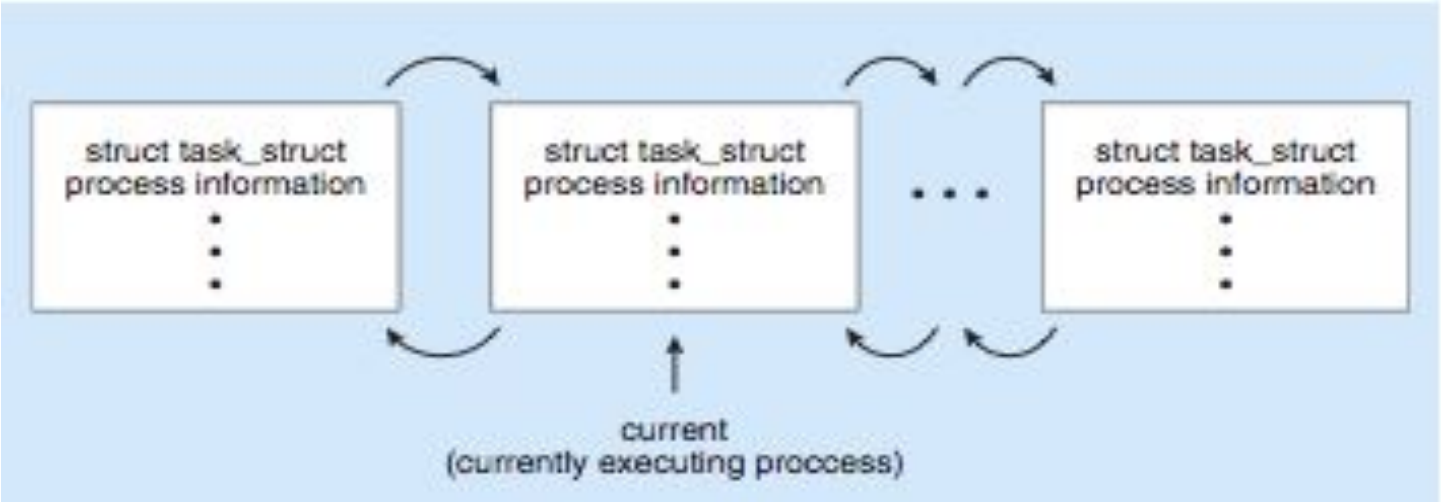
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process Representation in Linux

- Represented by the C structure `task_struct`

```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



Process Creation

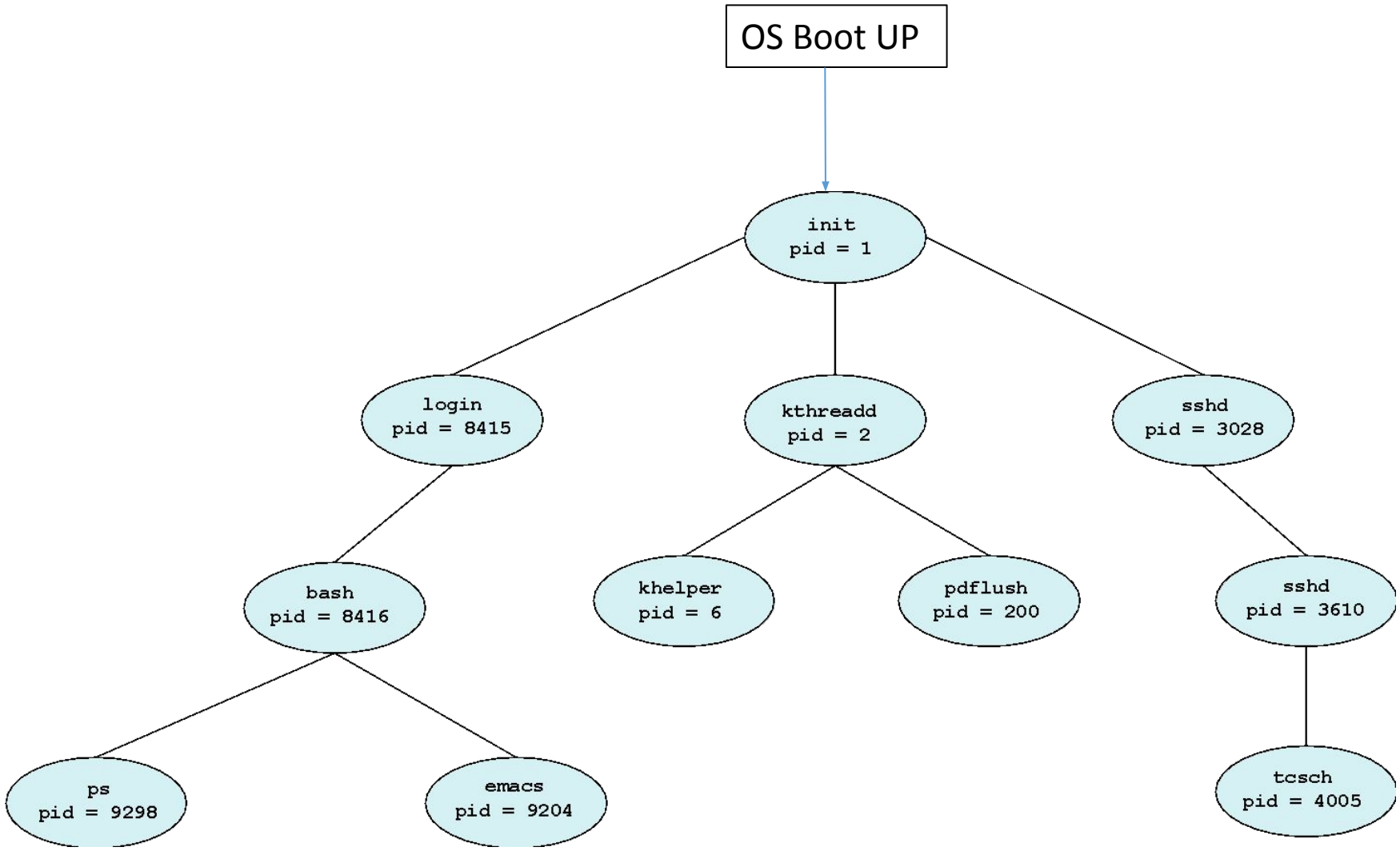
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- Generally, process identified and managed via a **process identifier** (**pid**)

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

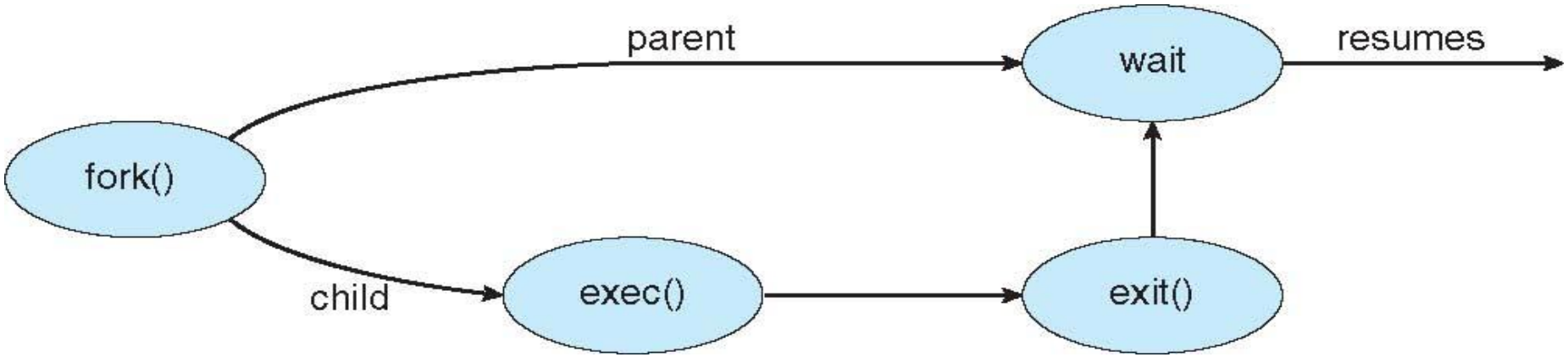
Processes Tree in Linux



Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Termination

- Process executes last statement and asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

- Wait for termination, returning the pid:

```
pid t pid; int status;
pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Assignment 1

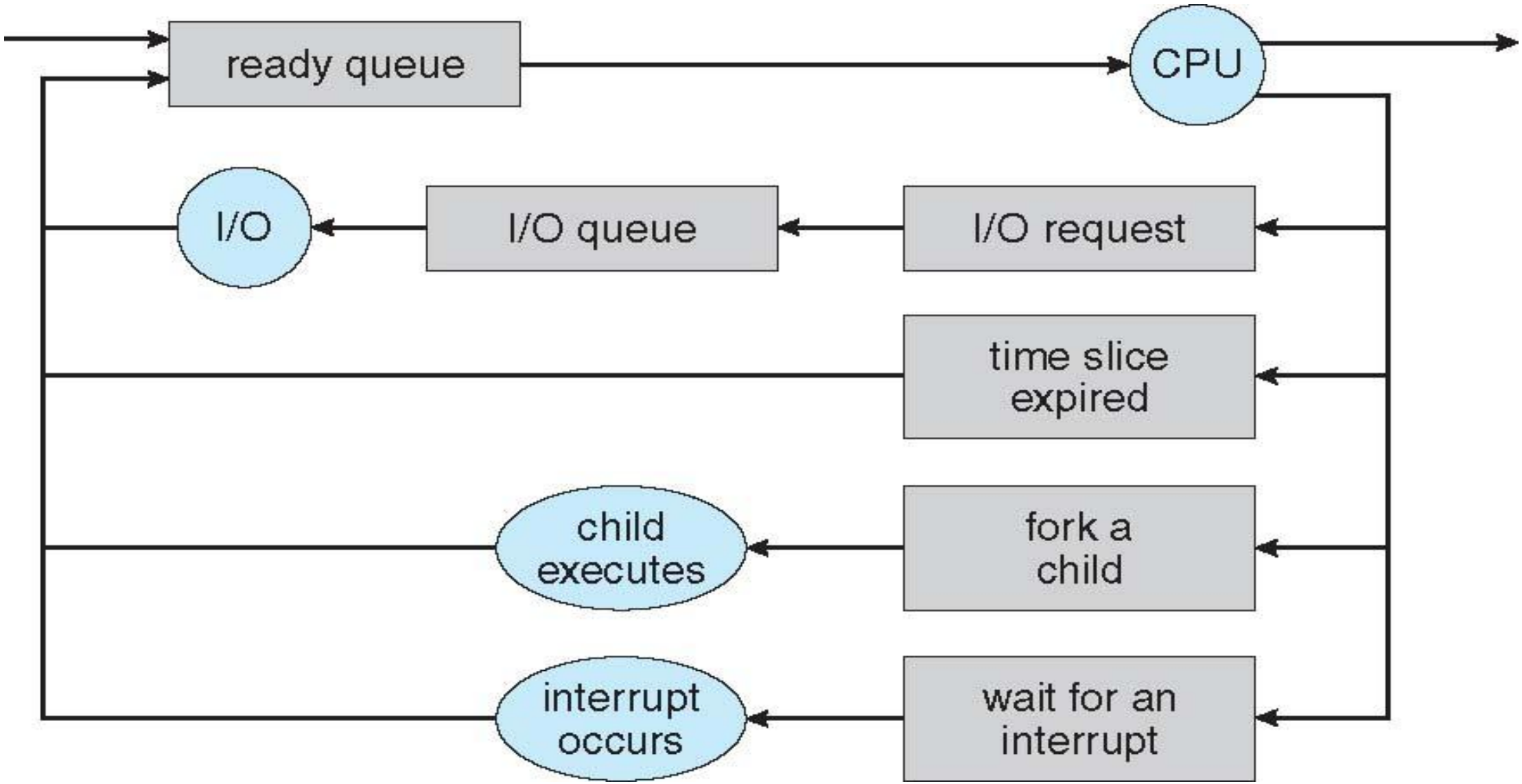
1.1 Learning how to use fork() and waitpid() system calls – multiple reads and writes to and from a file.

1.2 Basic (rudimentary) shell: Basic shell that parses user commands and executes internal or external commands.

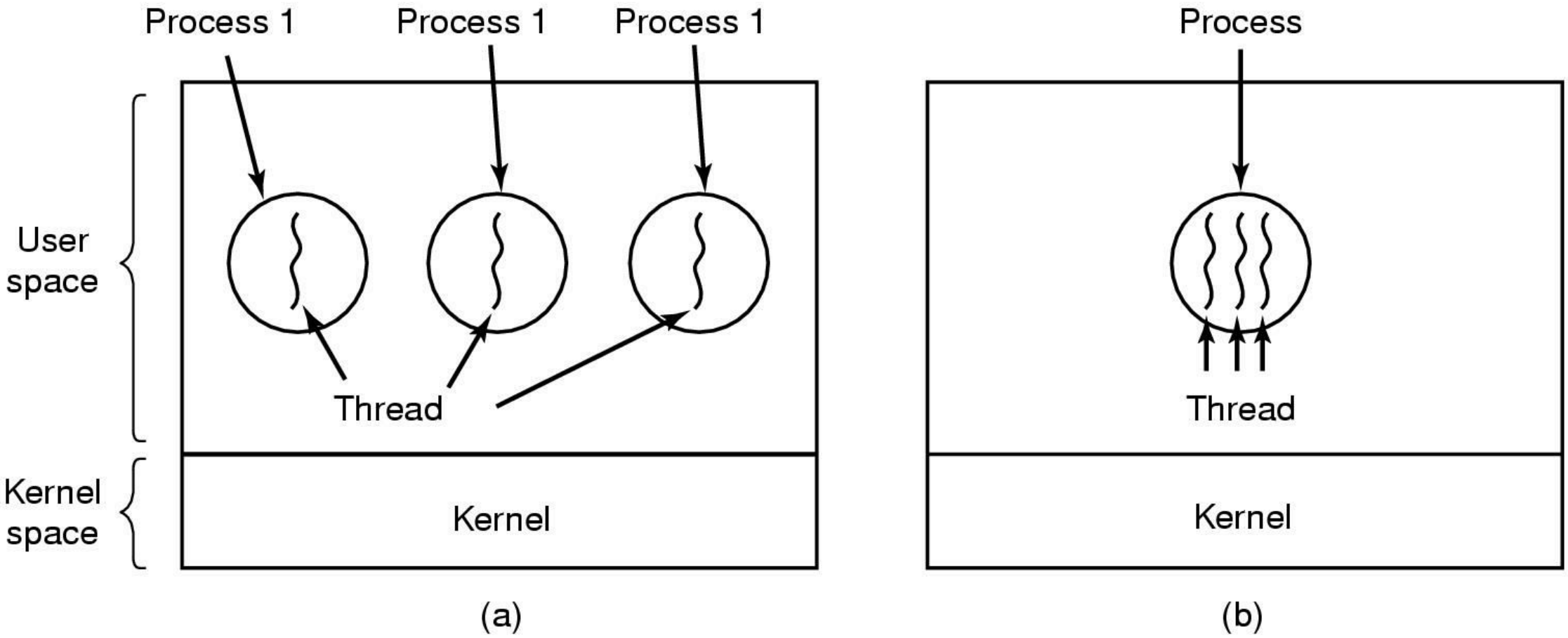
Internal commands: parsed and executed by the shell program (e.g. cd, pwd etc.)

External commands: correspond to individual programs that are run, each time a command is invoked. (e.g. vi, ls, cat etc.)

Process Queues



The Thread Model



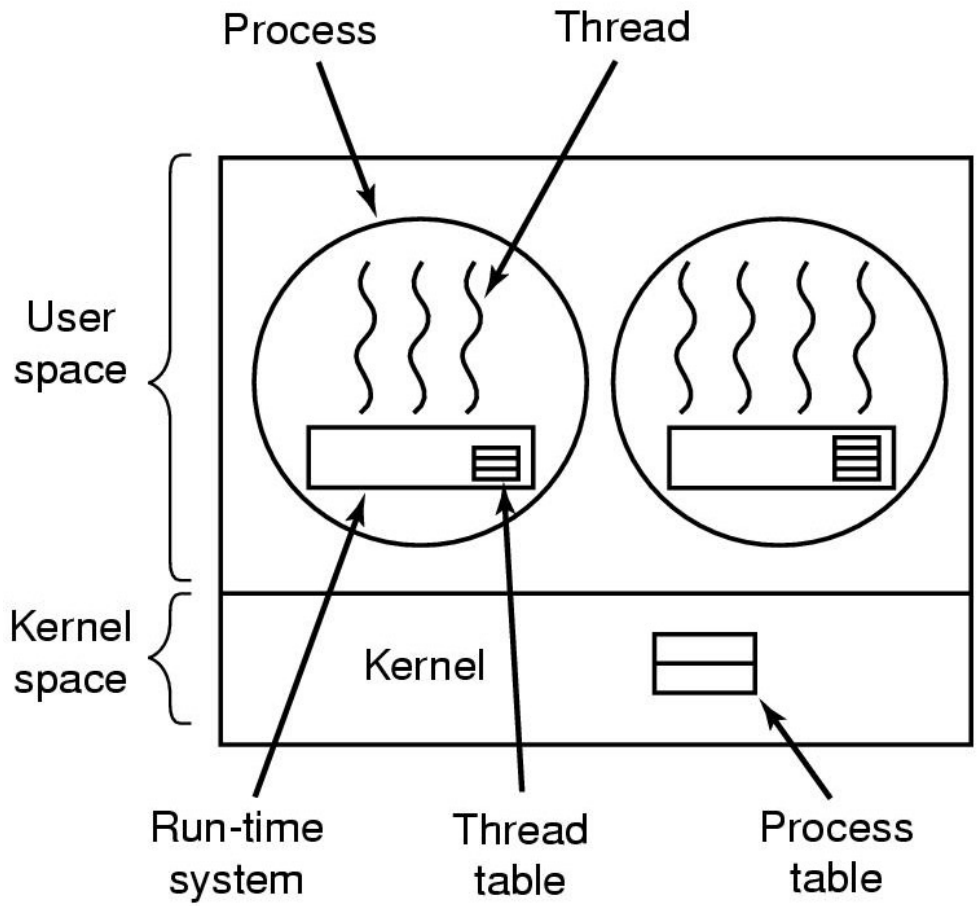
(a) Three processes each with one thread

(b) One process with three threads

Per process vs per thread items

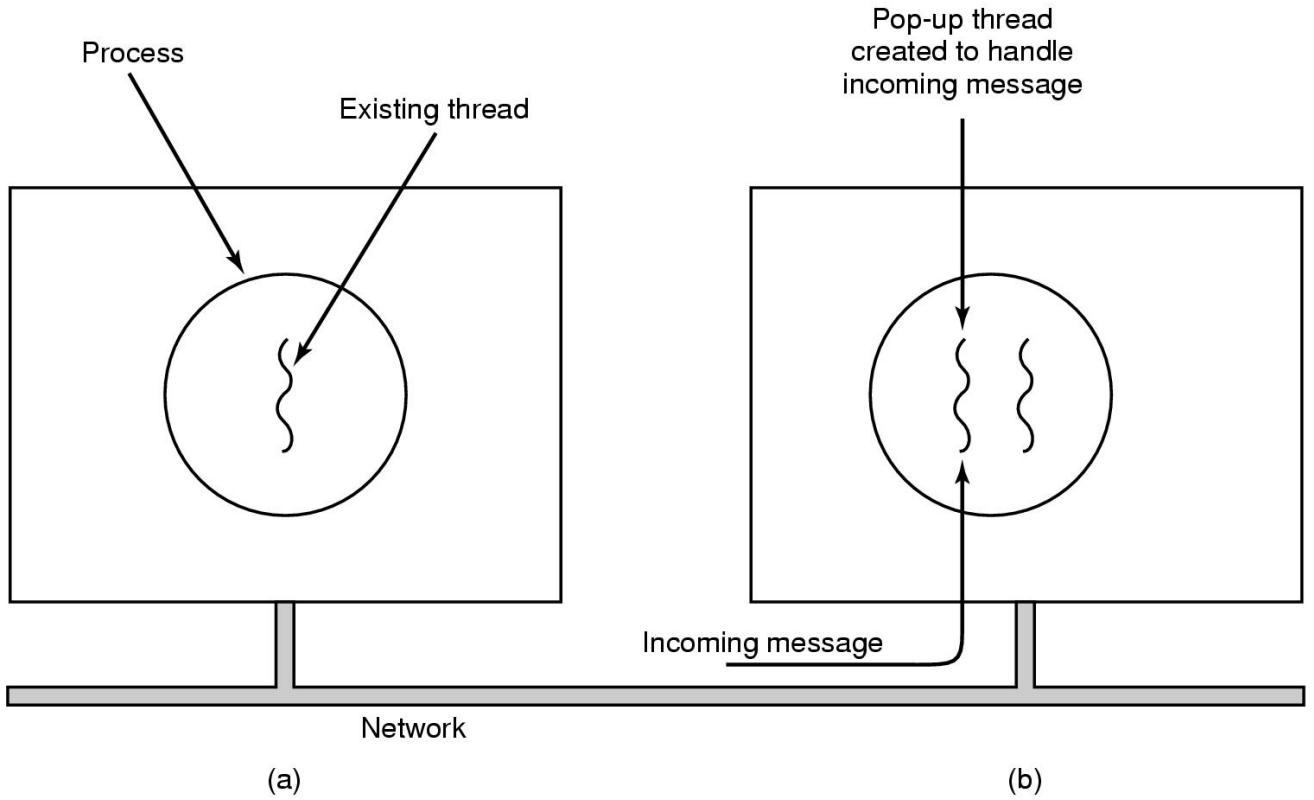
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Implementing Threads in User Space



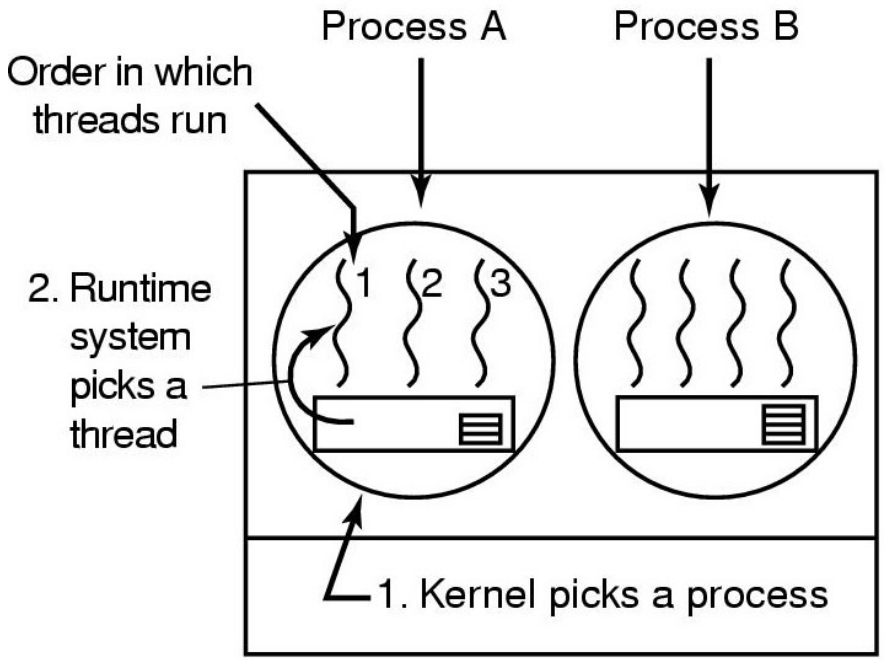
A user-level threads package

Pop-Up Threads



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

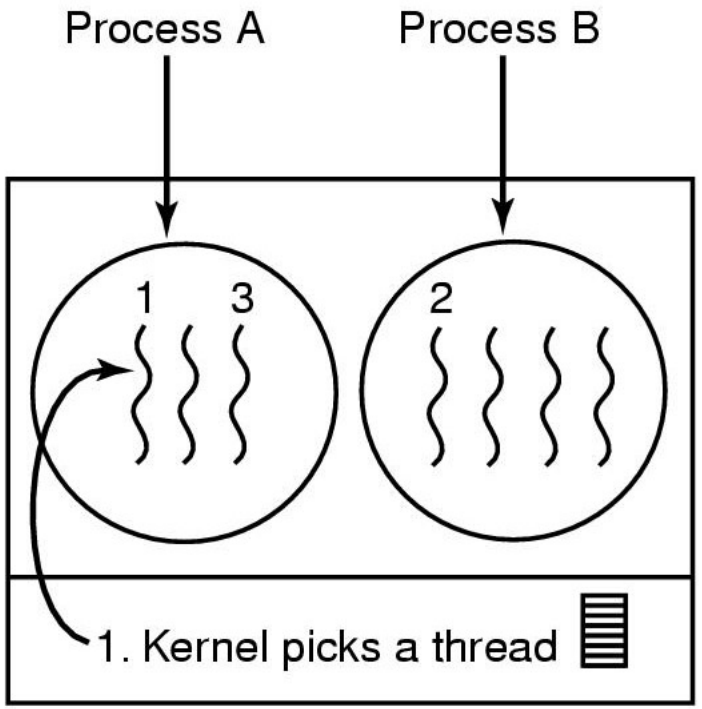
Thread Scheduling (1)



Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Thread Scheduling (2)



Possible: A1, A2, A3, ~~A1, A2, A3~~ B1, B2, B3

Also possible: A1, B1, A2, B2, A3, B3

Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Pthread Library

- Many thread models emerged: Solaris threads, win-32 threads
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Simply a collection of C functions.

Creating pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* func(void* arg)
{ pthread_detach(pthread_self());
printf("Inside the thread\n");
pthread_exit(NULL); }

void fun()
{   pthread_t ptid;
  pthread_create(&ptid, NULL, &func, NULL);
printf("This line may be printed before thread terminates\n");
if(pthread_equal(ptid, pthread_self())  printf("Threads are equal\n");
else printf("Threads are not equal\n");
pthread_join(ptid, NULL);
printf("This line will be printed after thread ends\n");
pthread_exit(NULL);
}
```

```
int main()
{
        fun();
        return 0;
}
```

Thread’s local data

- Variables declared within a thread (function) are called local data.
- Local (automatic) data associated with a thread are allocated on the stack. So these may be deallocated when a thread returns.
- So don’t plan on using locally declared variables for returning arguments. Plan to pass the arguments thru argument list passed from the caller or initiator of the thread.

Thread termination (destruction)

Implicit : Simply returning from the function executed by the thread terminates the thread. In this case thread's completion status is set to the return value.

- Explicit : Use thread_exit.

Prototype: void thread_exit(void *status);

The single pointer value in status is available to the threads waiting for this thread.

Waiting for thread exit

int pthread_join (pthread_t tid, void * *statusp);

Much like waitpid();