Processes, Threads & Scheduling

Using Tanenbaum's Modern Operating Systems (3rd edition)

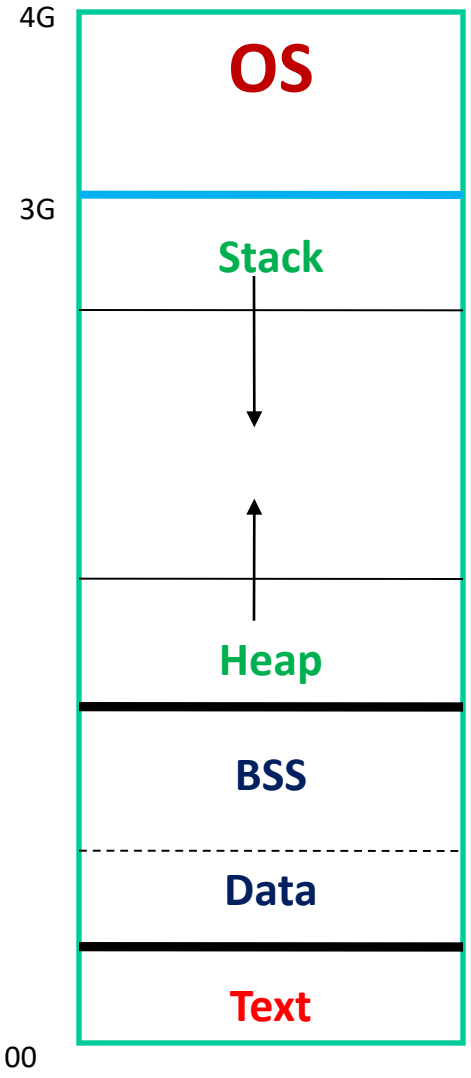
What is a Process?

- Instance of an executing program
- Includes its state: PC, registers, variables,…
- Each process has its own address space, which is protected from other processes by HW
- OS creates a sandbox for the process – makes it feel as if it “owns” the whole computer

Process Memory Map

E.g. UNIX Process has four segments:

- **Stack** – Local data
- **Heap** – User allocated data
- **BSS** – Uninitialized global data
- **Data** – Initialized global data
  - R/W area.
  - R/O area.
- **Text** – Program code



Pseudo-parallelism

- The CPU switches from program to program:
  - Running each for tens or hundreds of milliseconds
  - Giving an illusion of multiprogramming

- There is only one PC and set of registers:
  - When a process stops running, its environment is saved in the Process Control Block (PCB).
  - This environment is restored before the process resumes.

Multiprogramming

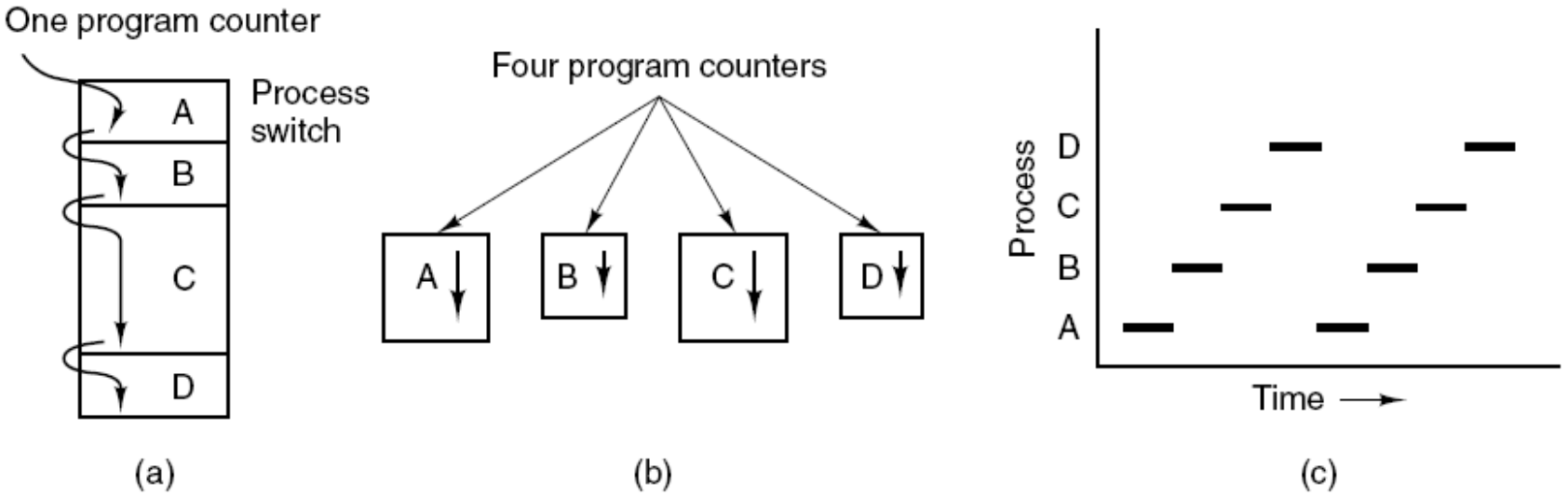


Figure 2-1.

a) Multiprogramming of four programs.
b) Conceptual model of four independent, sequential processes.
c) Only one program is active at once.

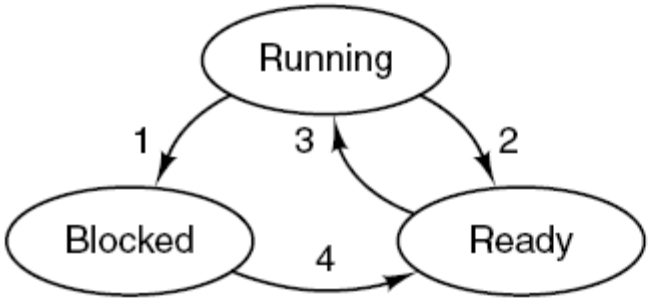
Time Sharing

- The **time-slice** method makes the process not to run uniformly, and often in an unpredictable scheduling.

- Programmer should not assume precise timing.

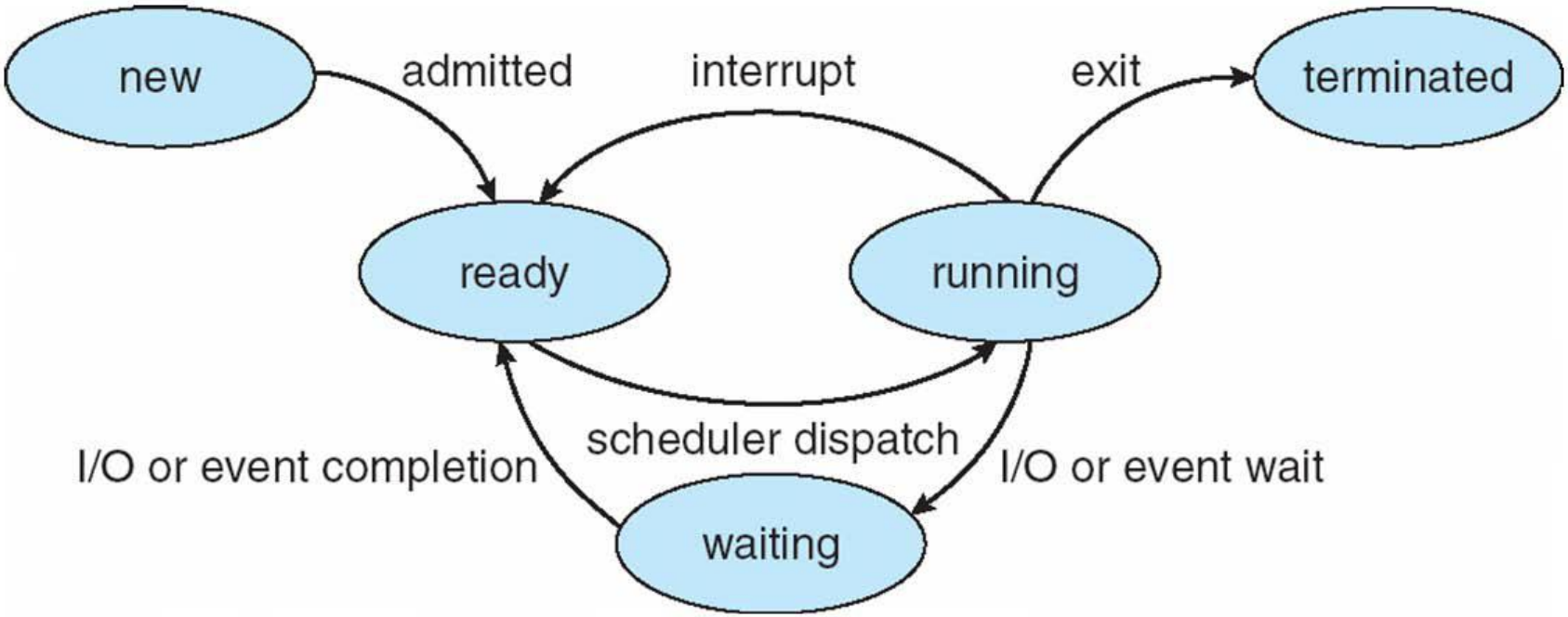
Process Life-cycle (simple)

- **Running** – actually using the CPU
- **Ready** – waiting for its turn of the CPU
- **Blocked** – waiting for something, unable to run until some external event has happened



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

Process Life-cycle (full)



Process Creation

- Several events cause process to be created:
  - System initialization
  - Execution of process creation system-call by running process
  - A user request

Process Creation (II)

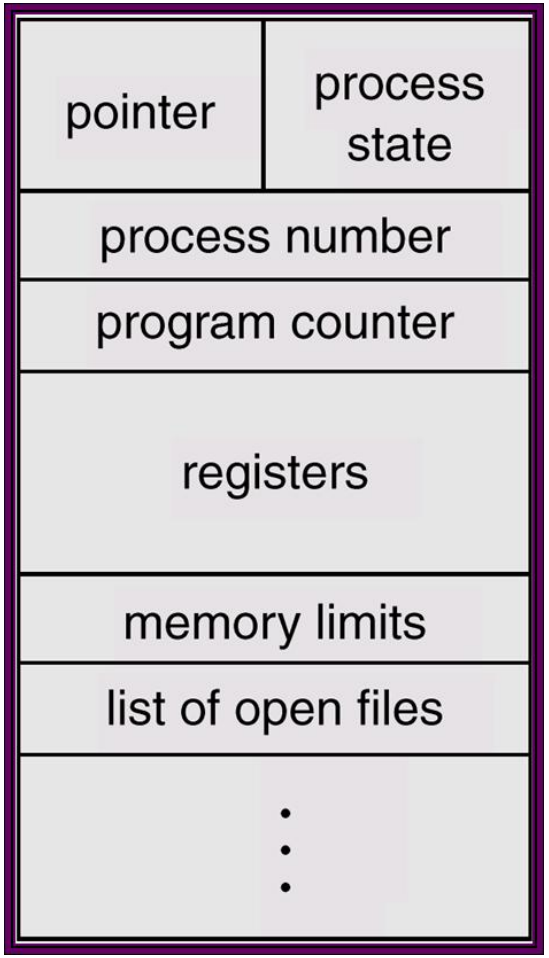
- UNIX – *fork()*:
  - The child inherits all the environment variables, memory image (variables) and resources (e.g. open files) of the parent
  - The returned value is used to determine if it is a parent or child.
  - The child may use another system call to load and run a different executable

- Windows – *CreateProcess()*:
  - Creates a new process, sets environment, loads and runs an executable in one go.

Process Termination

- Voluntary:
  - *exit()* system call in Unix and *ExitProcess()* in Windows
- Involuntary:
  - A process that makes a fatal error (e.g. division by zero) is signaled (interrupted) by the OS
  - By another: *kill()* in UNIX and *TerminateProcess()* in Windows

PCB – Process Control Block



Process Table

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		

Process Hierarchies

- UNIX
  - Process has only one parent
  - Each child may have children
  - Process cannot disinherit its child
- Windows
  - All processes are equal
  - The creator can pass a process handler to another process

The init process

- *init* process is the parent of all other processes
  - Creates processes from a script stored in the file */etc/inittab*
  - The *inittab* file describes which processes are started at boot-up and during normal operation

- *init* is invoked as the last step of the kernel boot sequence

Zombies

- Process collects statistics about itself (PID, termination status, resource usage information)
- This information is made available to the parent process – using *wait()* – after the process dies
- If this information is not collected, the entry stays in the kernel space and is called a zombie.
- Process must die if you send it a *"kill"* (-9).
- When a process dies, its child processes all become children of process number 1, whi0ch is the *init* process.
  - *init* is ``always'' waiting for children to die, so that they don't remain as zombies

Exercise #1 – Create Process

UsingFork

```
int g_var=0;
main() {
    int  l_var=0;

    n = fork();
    if (n<0){
         /* Error code handling */
    } else if (n>0) {
           /* Parent code */
           Loop: 1..30
              ++g_var; print g_var;
              ++l_var; print l_var;
              sleep(1);
    } else {
           /* Child code */
            Loop: 1..30
              --g_var; print g_var;
              --l_var; print l_var;
               sleep(1);
    }
}
```

The Shell Command

- The shell reads a command and forks a new process:

```
while(TRUE) {
        type_prompt();
        read_command(command, parameters);

        if (fork() != 0) {
                /* Parent Code */
                waitpid(-1, &status, 0);
        } else {
                /* Child code */
                execve(command,parameters);
        }
}
```

Homework + Interview Questions

- What is a “Process”?
- What are the different states of the Process
  - Explain the states and the transitions between
- How the process is created?
  - Explain the flow and the parent-child relationship
- Explain the terms:
  - PCB
  - Zombies
  - *init* process

Threads

Threads

- Process model based on 2 concepts:
  - Resource management: open-files, memory, accounting info, etc…
  - Execution
- Threads are executing in the context of a process using the same resources
- The ***pthreads*** library supports running several threads using the same process:

  status=pthread_create(threadID, …, FuncToRun, params)

Process vs. Thread

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	

Why threads instead of processes?

- The advantage of quasi-parallelism, but still sharing the same address space.
- Creating a thread takes 10-100 times faster than creating process.
- Multi-core – real parallelism.

Web Server – Threading Model (I)

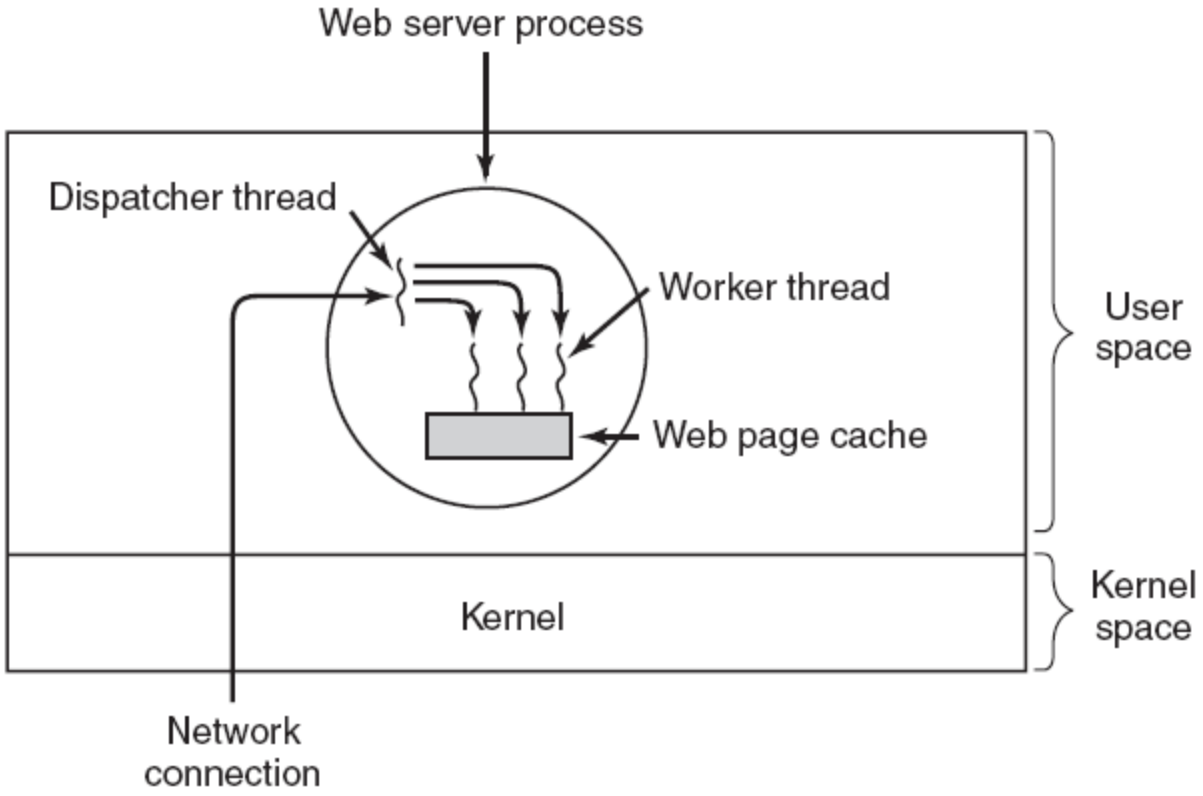


Figure 2-8. A multithreaded Web server.

Web Server – Threading Model (II)

The pseudo-code:

```
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)

Figure 2-9. A rough outline of the code for Fig. 2-8.
(a) Dispatcher thread.
(b) Worker thread.

Multi- vs. Single-Threaded

- Multi-Threaded process
  - Parallelism, blocking system calls
  - High performance, easy programming
- Single-Threaded process
  - No parallelism, blocking system calls
  - Low performance

POSIX Threads Functions

Thread Call	Description
pthread_create	Create thread
pthread_exit	Terminate 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 POSIX threads function calls

Exercise #2: Create Thread

```
PrintFunc(void* tid)
{
   /* Do something…*/
    pthread_exit();
}

int main()
{
   pthread_t threads[NUM_OF_THREADS];

  for (i .. NUM_OF_THREADS){
   /* Do something…*/
    status = pthread_create(&threads[i],…, PrintFunc, (void*) i);
   if (status)
       …
   }
}
```

Thread Implementation
User vs. Kernel Space

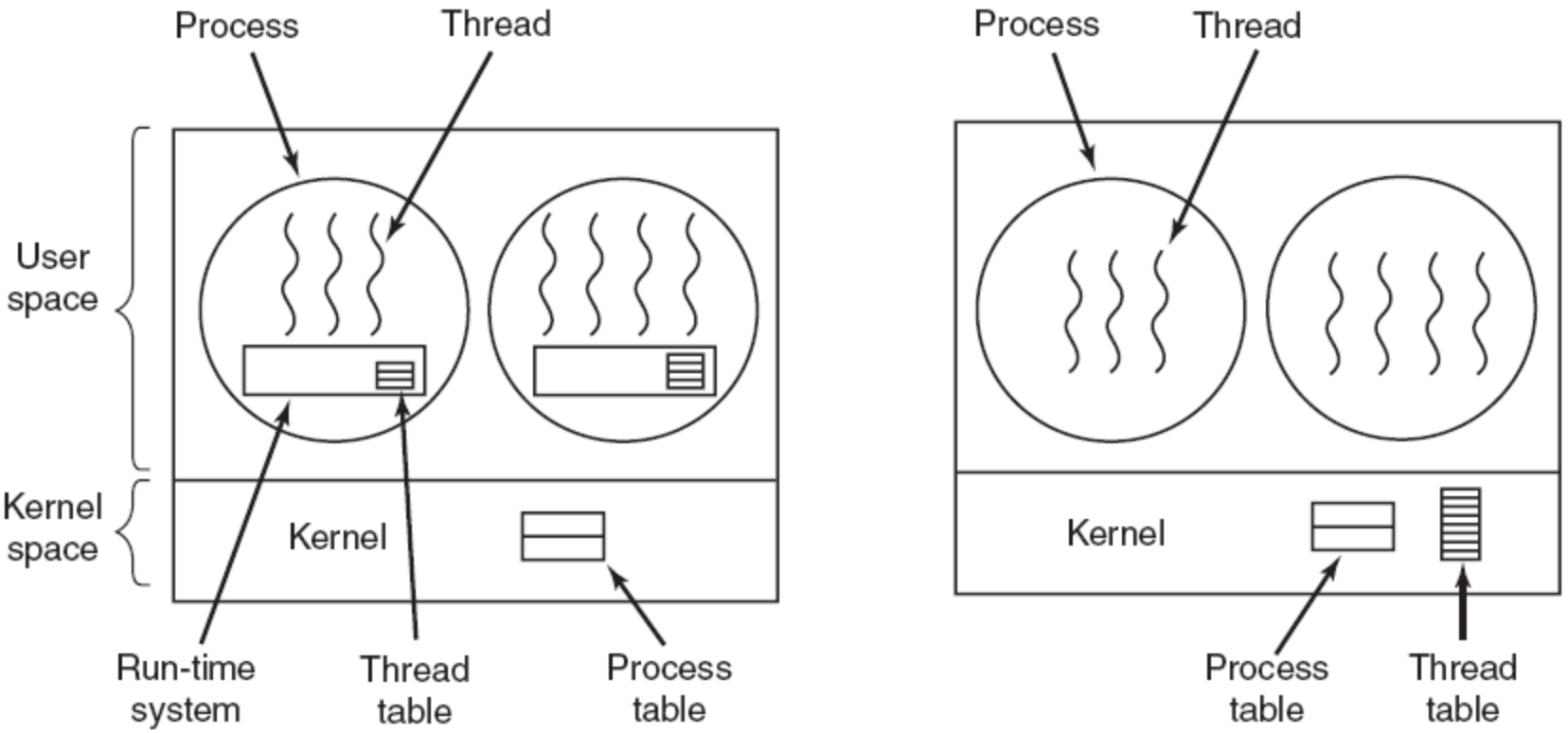


Figure 2-16. (a) A user-level threads package.
(b) A threads package managed by the kernel

User vs. Kernel (II)

- User mode/space implementation
  - ***Pros***
    Supports all OS, faster, scheduling algorithm per process.
  - ***Cons***
    Blocking system calls, page fault – shall stop the all threads, no clock interrupts
- Kernel mode/space implementation
  - ***Pros***
    Blocking system calls, page fault – shall be managed same as in process.
  - ***Cons***
    Not all OS, slower.

Pop-up Thread (I)

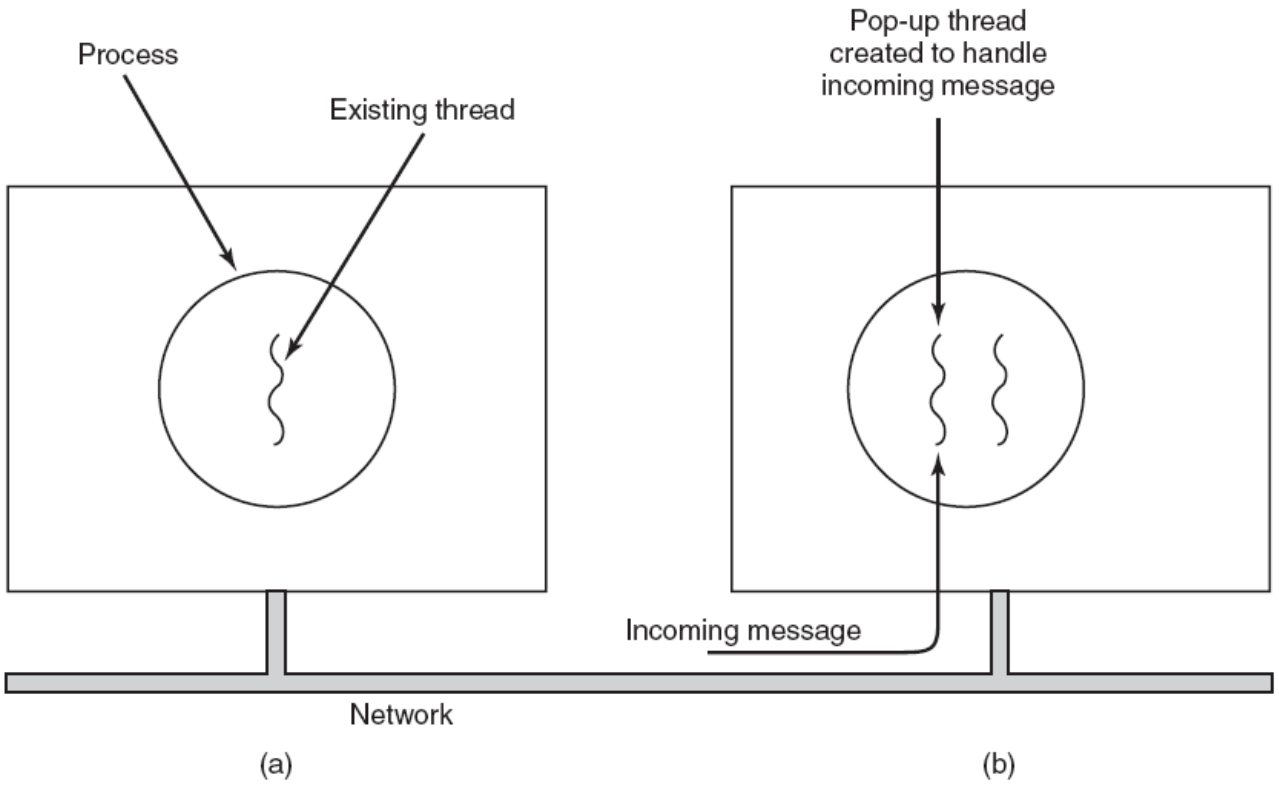


Figure 2-18. Creation of a new thread when a message arrives.
(a) Before the message arrives.
(b) After the message arrives.

Pop-up Thread (II)

- Arrival of a message causes the system to create a new thread
  - ***Pros***:
    Latency time between message arrival and processing is very short
  - ***Cons***
    Burst of incoming messages may cause system crash (Why?)

Multi-threaded code

Some pitfalls:

- Global variables corruption:

  Example: ***errno*** accessed by two threads calling system-calls

- Reentrant vs. Non-reentrant code:
  - Non-reentrant code was not designed to have a second call, made to any procedure, while a previous call has not yet finished

  Example: Call a procedure to assemble a message in a fixed buffer, before sending it over network.

Homework + Interview Questions

- What is *thread*?
- What is the difference between a *thread* and a *process*?
- What is the role of *pthread_join* function?
- Explain the terms:
  - Race condition
  - Reentrant code
  - Non-reentrant code
  - Critical section

Scheduler

Scheduler

- The Scheduler:
  - Is an integral part of the OS
  - Manages the running processes in the system
  - Runs a scheduling algorithm to force a required policy

- Three typical environments:
  - Batch (I/O bound operations – Banks, etc.)
  - User-Interactive (word-processor)
  - Real-time (voice-mail)

Typical Environments

- Different scheduling algorithms for different application types:
  - Batch – non preemptive, long intervals.
  - Interactive – preemptive, short intervals.
  - Real-time – preemptive , long intervals.

Scheduling Criteria

All systems

- Fairness – giving each process a fair share of the CPU
- Policy enforcement – seeing that stated policy is carried out
- Balance – keeping all parts of the system busy

Batch systems

- Throughput – maximize jobs per hour
- Turnaround time – minimize time between submission and termination
- CPU utilization – keep the CPU busy all the time

Interactive systems

- Response time – respond to requests quickly
- Proportionality – meet users' expectations

Real-time systems

- Meeting deadlines – avoid losing data
- Predictability – avoid quality degradation in multimedia systems

Scheduling Metrics

- **Throughput**
  - Jobs per hour.
- **Turnaround time**
  - Time from submission to completion.
- **Response time**
  - Time between submission and first response.
- **Proportionality**
  - Request type vs. response time.

- **CPU Utilization**
  - not a good metric.
  - 100% is using every cent of CPU but degrades performance.

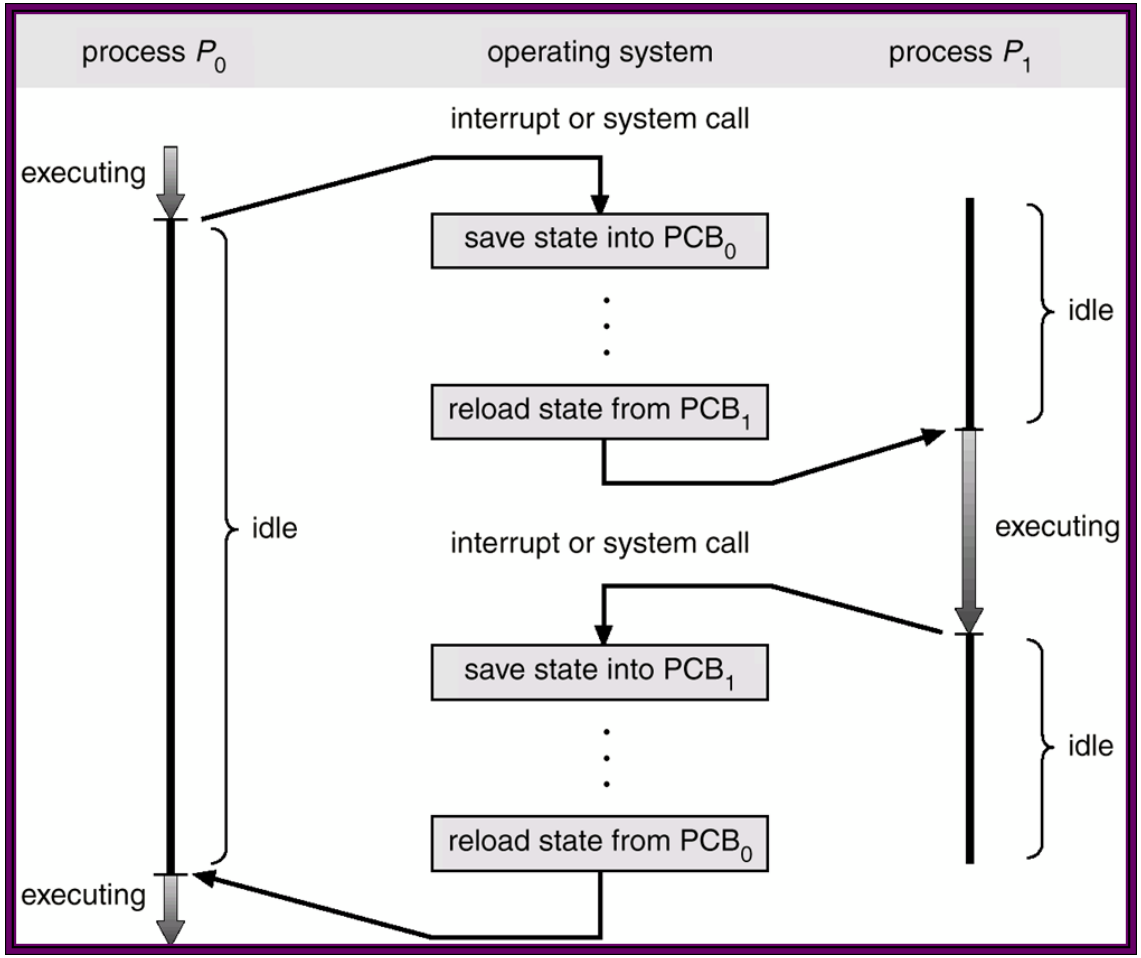
Preemptive vs. Non-preemptive

- **Preemptive** Scheduling:
  - Scheduler may suspend a running process to allow another processes to run.
  - Dangerous race conditions.
- **Non-preemptive** Scheduling:
  - Run each process to completion or blocking
  - Less race condition issues
  - Must prevent **starvation**

Dispatcher

- Gives CPU control to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

Context Switch (in practice)



Quantum

- Scheduler uses HW clock to implement the quantum. Adds itself to the interrupt vector of the clock
- Quantum must be shorter than mean CPU burst, otherwise performance will suffer
- Assuming context-switch takes 1 msec – reasonable quantum shall be 20-50 msec

First Come, First Served (FCFS)

- Non-preemptive.
- Treats ready queue as FIFO.
- Simple, but typically long/varying turnaround time:
  - CPU bound tasks.
  - I/O bound tasks – may take long time (why?).

Round Robin (RR)

- FCFS with Preemption
- Time quantum (or time-slice)
- Performance depends on quantum *q*:
  - Small *q*: overhead due to context switches & scheduling
  - Large *q*: Behaves like FCFS. Poor response time
  - *q* should be relatively fit to context-switching time

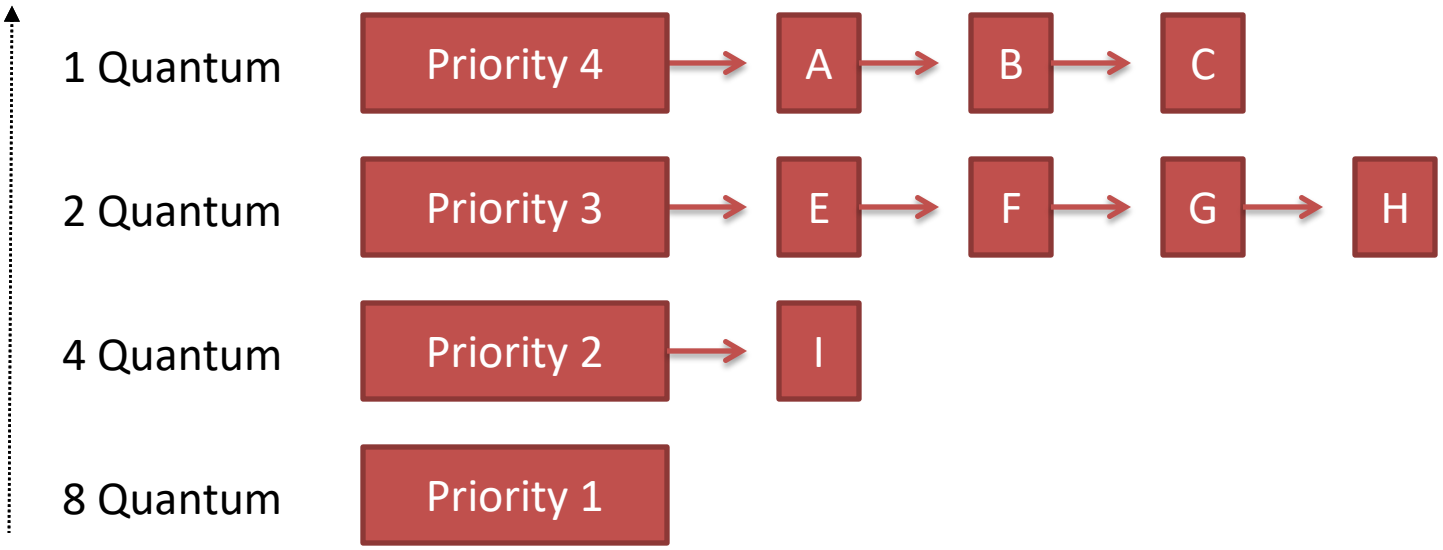
Priority Queue

- Priority associated with each process
- CPU is allocated to the process with highest priority; if equal, use FCFS
- **Problem**
  - Starvation (or Indefinite Blocking) – caused by fixed priority
- **Solution**
  - Dynamic priority.

Multilevel Queue

Avoid Starvation

Lowest priority (In UNIX 139)



Higher priority (In UNIX 0)

Priority Change

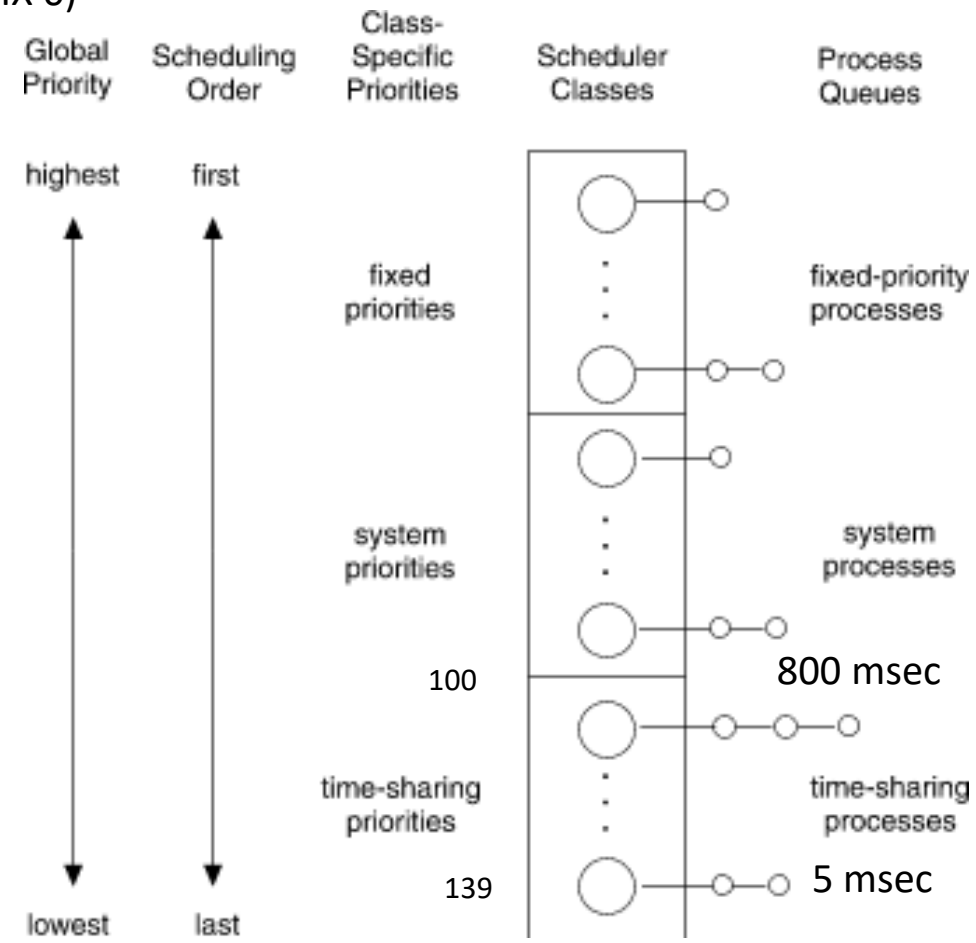
- Priority is changed by:
  - Decrease with every tick to prevent higher priority processes from hogging the CPU and starve lower priority processes
  - The ***`nice()`*** library-call on UNIX decreases priority to be nice to other processes
  - The system may assign priority to improve performance (i.e. giving higher priority to I/O-bound processes).

Scheduling Parameters

Process inherits its scheduling parameters, including scheduler class and a priority within that class

Default configuration – all user login shells, begin as time-sharing processes.

(In UNIX 0)



(In UNIX 139)

Win32

- Multilevel feedback queue, with 32 priority levels (0-31):
  - 16-31: Real-Time.
  - 1-15: User.
  - 0: Zero-page process
  - -1: System Idle Process
- Thread-based Scheduling
- Dynamic Priority
  - The kernel may change the priority level of a thread depending on its I/O and CPU usage and whether it is interactive (i.e. accepts and responds to input from humans)
  - Raising the priority of interactive and I/O bounded processes
  - Lowering that of CPU bound processes, to increase the responsiveness of interactive applications

Exercise #3: Process Priority

- Run your UsingFork program.
- Check the task priority using *"ps -al"* command.
- Re-run your program with different priority using *nice* command.

Homework + Interview Questions

- What is *context switch*? Explain how it works.
- Explains the roles & responsibilities of :
  - *Scheduler*
  - *Dispatcher*
- Explain the terms:
  - *PCB*
  - *Round Robin*
  - *Preemptive scheduling*
  - *Quantum*
  - *Static vs. Dynamic Priorities*