Processes

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- Process Concept
- Implementing Processes
- Context Switch
- Process Creation APIs
Process Concept

What is the process?

- An instance of a program in execution.
- An encapsulation of the flow of control in a program.
- A dynamic and active entity.
- The basic unit of execution and scheduling.
- A process is named using its process ID (PID).
- Job, task, or sequential process
- A process includes:
  - CPU contexts (registers)
  - OS resources (memory, open files, etc.)
  - Other information (PID, state, owner, etc.)
Process in Memory

- **read-only segment** (.init, .text, .rodata)
- **read/write segment** (.data, .bss)
- **run-time heap** (managed by malloc)
- **user stack** (created at runtime)
- **kernel virtual memory** (code, data, heap, stack)

Memory invisible to user code:
- Stack pointer
- brk

Diagram:
- Memory structure
- Program
- Code and data
Running a Process

Source: Jin-Soo Kim (Sungkyunkwan Univ.)
Running Multiple Processes

Source: Jin-Soo Kim (Sungkyunkwan Univ.)
Interleaving Multiple Processes

Source: Jin-Soo Kim (Sungkyunkwan Univ.)
Virtualizing the CPU

OS creates the illusion that each process has its own CPU (and memory)

Source: Jin-Soo Kim (Sungkyunkwan Univ.)
Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
Process State – Linux Example

- **R:** Runnable
- **S:** Sleeping
- **T:** Traced or Stopped
- **D:** Uninterruptible Sleep
- **Z:** Zombie
- **<:** High-priority task
- **N:** Low-priority task
- **S:** Session leader
- **+:** In the foreground process group
- **I:** Multi-threaded
Process Creation (1)

- Process hierarchy
  - One process can create another process: parent-child relationship
  - UNIX calls the hierarchy a "process group"
  - Windows has no concept of process hierarchy.
  - Browsing a list of processes:
    - `ps` in UNIX
    - `taskmgr` (Task Manager) in Windows

```
$ cat file1 | wc
```
Process Creation (2)

- Process creation events
  - Calling a system call
    - `fork()` in POSIX, `CreateProcess()` in Win32
    - Shells or GUIs use this system call internally.
  - System initialization
    - `init` process

- Background processes
  - Do not interact with users
  - Daemons
Process Creation (3)

- **Resource sharing**
  - Parent may inherit all or a part of resources and privileges for its children
    - UNIX: User ID, open files, etc.

- **Execution**
  - Parent may either wait for it to finish, or it may continue in parallel.

- **Address space**
  - Child duplicates the parent’s address space or has a program loaded into it.
Process Termination (1)

- Process termination events
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
    - Exceed allocated resources
    - Segmentation fault
    - Protection fault, etc.
  - Killed by another process (involuntary)
    - By receiving a signal
Process Termination (2)

- Process executes last statement and then asks the kernel to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process’ resources are deallocated by the kernel

- Parent process may wait for termination of a child process
  - \texttt{pid = wait(&status);}
  - Returns pid and termination status of a child

- If no parent waiting (did not invoke wait())
  - Process is a zombie

- If parent terminated without invoking wait
  - Process is an orphan
Diagram of Process State

- **new**
- **admitted**
- **interrupt**
- **exit**
- **terminated**

States:
- **ready**
- **running**
- **waiting**

Transitions:
- **I/O or event completion**
- **scheduler dispatch**
- **I/O or event wait**
Process Control Block (PCB)

- PCB (Process Control Block)
  - Each PCB represents a process
  - Also called task control block
  - Contains all of the information about a process
    - Process state
    - Program counter
    - CPU registers
    - CPU scheduling information
    - Memory-management information
    - Accounting information
    - I/O status information, etc.
  - task_struct in Linux
    - 2584 bytes as of Linux 4.1.4
PCBs and Hardware State

- **When a process is running:**
  - Its hardware state is inside the CPU: PC, SP, registers

- **When the OS stops running a process:**
  - It saves the registers’ values in the PCB.

- **When the OS puts the process in the running state:**
  - It loads the hardware registers from the values in that process’ PCB.
Context Switch

- Context switch (or process switch)
  - The act of switching the CPU from one process to another.
  - Administrative overhead
    - saving and loading registers and memory maps
    - flushing and reloading the memory cache
    - updating various tables and lists, etc.
  - Context switch overhead is dependent on hardware support.
    - Multiple register sets in UltraSPARC.
    - Advanced memory management techniques may require extra data to be switched with each context.
  - Context-switch time is overhead
    - The system does no useful work while switching
    - The more complex the OS and the PCB \(\Rightarrow\) the longer the context switch
  - 100s or 1000s of switches/s typically.
**Context Switch – Linux Example**

- **Linux example**
  - Total 1,693,515,228 ticks = 4704 hours = 196 days
  - Total 4,066,419,922 context switches
  - Roughly 240 context switches / sec
Context Switch between Processes

process $P_0$ operating system process $P_1$

executing

interrupt or system call

save state into PCB$_0$

idle

save state into PCB$_1$

executing

interrupt or system call

reload state from PCB$_1$

idle

reload state from PCB$_0$
**Process State Queues (1)**

- **State queues**
  - The OS maintains a collection of queues that represent the state of all processes in the system
    - *Job queue* – all processes in the system
    - *Ready queue* – all processes ready to execute
    - *Wait queues* – processes waiting for a particular event
      - (e.g., disk, timer, resources, mutex, …)
  - Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is migrated between the various queues.
Process State Queues (2)

The diagram illustrates the structure of process state queues with various units such as queue header, PCBs, and other components. The diagram shows the connections between the different elements, indicating the flow and organization of processes within these queues.
Queueing diagram represents queues, resources, flows

- ready queue
- I/O
- I/O queue
- I/O request
- time slice expired
- child executes
- fork a child
- interrupt occurs
- wait for an interrupt
Process State Queues (4)

- PCBs and state queues
  - PCBs are data structures
    - dynamically allocated inside OS memory
  - When a process is created:
    - OS allocates a PCB for it
    - OS initializes PCB
    - OS puts PCB on the correct queue
  - As a process computes:
    - OS moves its PCB from queue to queue
  - When a process is terminated:
    - OS deallocates its PCB
### fork()

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Initializes the address space with a copy of the entire contents of the address space of the parent.
- Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Places the PCB on the ready queue.
- Returns the child’s PID to the parent, and zero to the child.
#include <sys/types.h>
#include <unistd.h>

int main()
{
  int pid;

  if ((pid = fork()) == 0)
    /* child */
    printf ("Child of %d is %d\n", getppid(), getpid());
  else
    /* parent */
    printf ("I am %d. My child is %d\n", getpid(), pid);
}
fork(): Example Output

% ./a.out
I am 31098. My child is 31099.
Child of 31098 is 31099.

% ./a.out
Child of 31100 is 31101.
I am 31100. My child is 31101.
Process Creation: UNIX (2)

exec()

- Stops the current process
- Loads the program "prog" into the process’ address space.
- Initializes hardware context and args for the new program.
- Places the PCB on the ready queue.
  – Note: exec() does not create a new process.
- What does it mean for exec() to return?

```c
int exec (char *prog, char *argv[])
```
Simplified UNIX Shell

```c
int main()
{
    while (1) {
        char *cmd = read_command();
        int pid;
        if ((pid = fork()) == 0) {
            /* Manipulate stdin/stdout/stderr for pipes and redirections, etc. */
            exec(cmd);
            panic("exec failed!");
        } else {
            wait (pid);
        }
    }
}
```
Process Creation: NT

BOOL CreateProcess (char *prog, char *args, ...)

- **CreateProcess()**
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by "prog" into the address space
  - Copies "args" into memory allocated in address space
  - Initializes the hardware context to start execution at main
  - Places the PCB on the ready queue
Why fork()?

- Very useful when the child...
  - is cooperating with the parent.
  - relies upon the parent’s data to accomplish its task.
  - Example: Web server

```c
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```
Inter-Process Communications

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need inter-process communication (IPC)
  - Shared memory
  - Message passing
Communications Models

(a) Message passing. (b) shared memory.
Inter-Process Communications

- **Inside a machine**
  - Pipe
  - FIFO
  - Shared memory
  - Sockets

- **Across machines**
  - Sockets
  - RPCs (Remote Procedure Calls)
  - Java RMI (Remote Method Invocation)
Threads

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- Why threads?
- Thread interface
- Threading issues
- How to implement threads?
  - User-level threads
  - Kernel-level threads
- Threading models
Processes

- Heavy-weight
  - A process includes many things:
    - An address space (all the code and data pages)
    - OS resources (e.g., open files) and accounting info.
    - Hardware execution state (PC, SP, registers, etc.)
  - Creating a new process is costly because all of the data structures must be allocated and initialized
    - Linux: over 100 fields in task_struct
      (excluding page tables, etc.)
  - Inter-process communication is costly, since it must usually go through the OS
    - Overhead of system calls and copying data
Web server example

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task.

```c
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```
Cooperating Processes

- **Example**
  - A web server, which forks off copies of itself to handle multiple simultaneous tasks
  - Any parallel program on a multiprocessor

- **We need to:**
  - Create several processes that execute in parallel
  - Cause each to map the same address space to share data (e.g., shared memory)
  - Have the OS schedule these processes in parallel

- **This is very inefficient!**
  - Space: PCB, page tables, etc.
  - Time: creating OS structures, fork and copy address space, etc.
Rethinking Processes

What’s similar in these cooperating processes?

• They all share the same code and data (address space)
• They all share the same privilege
• They all share the same resources (files, sockets, etc.)

What’s different?

• Each has its own hardware execution state: PC, registers, SP, and stack.
Key Idea (1)

- Separate the concept of a process from its execution state
  - Process: address space, resources, other general process attributes (e.g., privileges)
  - Execution state: PC, SP, registers, etc.

- This execution state is usually called
  - a thread of control,
  - a thread, or
  - a lightweight process (LWP)
Key Idea (2)

(a) User space
   - Process 1
     - Thread
   - Kernel

(b) Process
   - Thread
   - Kernel
Key Idea (3)

![Diagram showing single-threaded and multithreaded processes with registers, stack, code, data, and files.](image)

- **Single-threaded process**
  - Registers
  - Stack
  - Thread
  - Code
  - Data
  - Files

- **Multithreaded process**
  - Registers
  - Registers
  - Registers
  - Stack
  - Stack
  - Stack
  - Thread
What is a Thread?

- **A thread of control (or a thread)**
  - A sequence of instructions being executed in a program.
  - Usually consists of
    - a program counter (PC)
    - a stack to keep track of local variables and return addresses
    - registers
  - Threads share the process instructions and most of its data.
    - A change in shared data by one thread can be seen by the other threads in the process
  - Threads also share most of the OS state of a process.
Using threads

- We can create a new thread for each request.

```c
webserver ()
{
    While (1) {
        int sock = accept();
        thread_fork (handle_request, sock);
    }
}
handle_request (int sock)
{
    /* Process request */
    close (sock);
}```
Concurrent Servers: Threads

1. Request
2. Create new thread to service the request
3. Resume listening for additional client requests
Multithreading

- **Benefits**
  - Creating concurrency is cheap.
  - Improves program structure.
  - Throughput
    - By overlapping computation with I/O operations
  - Responsiveness (User interface / Server)
    - Can handle concurrent events (e.g., web servers)
  - Resource sharing
  - Economy
  - Utilization of multiprocessor architectures
    - Allows building parallel programs.
Processes vs. Threads

- A thread is bound to a single process.
- A process, however, can have multiple threads.
- Sharing data between threads is cheap: all see the same address space.
- Threads become the unit of scheduling.
- Processes are now containers in which threads execute.
- Processes become static, threads are the dynamic entities.
Process Address Space

0xFFFFFFFF

address space

0x00000000

stack
(dynamically allocated mem)

heap
(dynamically allocated mem)

static data
(data segment)

code
(text segment)

PC

SP
Address Space with Threads

0xFFFF_FFF_FF

address space

thread 1 stack

thread 2 stack

thread 3 stack

heap
(dynamically allocated mem)

static data
(data segment)

code
(text segment)

SP (T1)

SP (T2)

SP (T3)

PC (T2)

PC (T1)

PC (T3)

0xFFFF_FFF_FF

0x00000000
<table>
<thead>
<tr>
<th># threads per addr space:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Early Macintosh</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
<td>Many embedded Oses</td>
<td>Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(VxWorks, uClinux, ..)</td>
<td></td>
</tr>
</tbody>
</table>
Threads Interface (1)

- **Pthreads**
  - 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.
  - Common in UNIX operating systems.
Threads Interface (2)

- POSIX-style threads
  - Pthreads
  - DCE threads (early version of Pthreads)
  - Unix International (UI) threads (Solaris threads)
    - Sun Solaris 2, SCO Unixware 2

- Microsoft-style threads
  - Win32 threads
    - Microsoft Windows 98/NT/2000/XP
  - OS/2 threads
    - IBM OS/2
**Pthreads (1)**

- Thread creation/termination

```c
int pthread_create (pthread_t *tid,
                    pthread_attr_t *attr,
                    void *(start_routine)(void *),
                    void *arg);

void pthread_exit (void *retval);

int pthread_join (pthread_t tid,
                   void **thread_return);
```
**Pthreads (2)**

- **Mutexes**

```c
int pthread_mutex_init
    (pthread_mutex_t *mutex,
     const pthread_mutexattr_t *mattr);

void pthread_mutex_destroy
    (pthread_mutex_t *mutex);

void pthread_mutex_lock
    (pthread_mutex_t *mutex);

void pthread_mutex_unlock
    (pthread_mutex_t *mutex);
```
Pthreads (3)

- Condition variables

```c
int pthread_cond_init
    (pthread_cond_t *cond,
     const pthread_condattr_t *cattr);

void pthread_cond_destroy
    (pthread_cond_t *cond);

void pthread_cond_wait
    (pthread_cond_t *cond,
     pthread_mutex_t *mutex);

void pthread_cond_signal
    (pthread_cond_t *cond);

void pthread_cond_broadcast
    (pthread_cond_t *cond);
```
Kernel/User-level Threads

**Who is responsible for creating/managing threads?**

- The OS (kernel threads)
  - Thread creation and management requires system calls
- The user-level process (user-level threads)
  - A library linked into the program manages the threads

**Why is user-level thread management possible?**

- Threads share the same address space
  - The thread manager doesn’t need to manipulate address spaces
- Threads only differ in hardware contexts (roughly)
  - PC, SP, registers
  - These can be manipulated by the user-level process itself.
Kernel-level Threads (1)

- **OS-managed threads**
  - The OS manages threads and processes.
  - All thread operations are implemented in the kernel.
  - The OS schedules all of the threads in a system.
    - If one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process.
    - Possible to overlap I/O and computation inside a process.
  - Kernel threads are cheaper than processes.
    - Less state to allocate and initialize
  - Windows 98/NT/2000/XP/Vista, Solaris, Tru64 Unix, Linux, Mac OS X
Implementing Kernel-level Threads

- **Kernel-level threads**
  - Kernel-level threads are similar to original process management and implementation.
Kernel-level Threads (2)

- Limitations

  - They can still be too expensive.
    - For fine-grained concurrency, we need even cheaper threads.
    - Ideally, we want thread operations as fast as a procedure call.
  - Thread operations are all system calls.
    - The program must cross an extra protection boundary on every thread operation, even when the processor is being switched between threads in the same address space.
    - The OS must perform all of the usual argument checks.
  - Must maintain kernel state for each thread.
    - Can place limit on the number of simultaneous threads. (typically ~1000)
  - Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.
User-level Threads (1)

- **Motivation**
  - To make threads cheap and fast, they need to be implemented at the user level.
  - Portable: User-level threads are managed entirely by the runtime system (user-level library).

- **User-level threads are small and fast**
  - Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB).
  - Creating a thread, switching between threads, and synchronizing threads are done via procedure calls (No kernel involvement).
  - User-level thread operations can be 10-100x faster than kernel-level threads.
Implementing User-level Threads (1)

- User-level threads
User-level Threads (2)

- Limitations
  - User-level threads are invisible to the OS.
    - They are not well integrated with the OS
  - As a result, the OS can make poor decisions.
    - Scheduling a process with only idle threads
    - Blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run.
    - Unscheduling a process with a thread holding a lock.
  - Solving this requires coordination between the kernel and the user-level thread manager.
    - e.g., all blocking system calls should be emulated in the library via non-blocking calls to the kernel.
Thread context switch

- Very simple for user-level threads
  - Save context of currently running thread
    : push all machine state onto its stack
  - restore context of the next thread
    : pop machine state from next thread’s stack
  - the next thread becomes the current thread
  - return to caller as the new thread
    : execution resumes at PC of next thread

- All done by assembly languages
  - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls.
Implementing User-level Threads (3)

- **Thread scheduling**
  - A thread scheduler determines when a thread runs.
    - Just like the OS and processes
    - But implemented at user-level as a library
  - It uses queues to keep track of what threads are doing.
    - Run queue: threads currently running
    - Ready queue: threads ready to run
    - Wait queue: threads blocked for some reason
      (maybe blocked on I/O or a lock)
  - How can we prevent a thread from hogging the CPU?
    - Periodic interrupt by OS (e.g., SIGALRM)
CPU Scheduling

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multi-processor Scheduling
- Operating Systems Examples
CPU Scheduling

- CPU scheduling
  - Deciding which process to run next, given a set of runnable processes.
  - Happens frequently, hence should be fast.

- Scheduling points
Schedulers (1)

- **Short-term scheduler (or CPU scheduler)**
  - Selects which process should be executed next
  - Sometimes the only scheduler in a system
  - Invoked frequently (milliseconds)

- **Long-term scheduler (or job scheduler)**
  - Selects which processes should be brought into the ready queue
  - Invoked infrequently (seconds, minutes)
  - Controls the degree of multiprogramming

- **Long-term scheduler strives for good process mix**
Schedulers (2)

- **Medium-term scheduler**
  - To decrease the degree of multiple programming due to resource shortage
  - **Swapping**
    - Remove process from memory
    - Store on disk
    - Bring back in from disk to continue execution
CPU Scheduling: Basic Concepts

- To maximize CPU utilization in multiprogramming
- Process execution consists of
  - CPU execution
  - I/O wait

```
load store
add store
read from file

CPU burst

store increment
index
write to file

I/O burst

CPU burst

load store
add store
read from file

I/O burst

wait for I/O

I/O burst

CPU burst
```
Execution Characteristics (1)

- Frequency vs. Burst Duration (milliseconds)

- The graph shows the frequency of bursts as a function of their duration, indicating a sharp decline in frequency as the burst duration increases.
### Execution Characteristics (2)

- **CPU burst vs. I/O burst**
  - A CPU-bound process
  - An I/O-bound process

![Diagram](image)
CPU Scheduling (1)

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways

- **CPU scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

- **Scheduling under 1 and 4 is nonpreemptive**
- **All other scheduling is preemptive**
CPU Scheduling (2)

- **Non-preemptive scheduling**
  - The scheduler waits for the running job to voluntarily yield the CPU.
  - Jobs should be cooperative.

- **Preemptive scheduling**
  - The scheduler can interrupt a job and force a context switch.
  - What happens
    - If a process is preempted in the midst of updating the shared data?
    - If a process in a system call is preempted?
CPU Scheduling (3)

- **Scheduling Criteria**
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – # of processes that complete their execution per time unit
  - **Turnaround time** – amount of time from process arrival to completion
  - **Waiting time** – amount of time a process has been waiting in the ready queue
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
CPU Scheduling (4)

- **Starvation**
  - A situation where a process is prevented from making progress because another process has the resource it requires.
    - Resource could be the CPU or a lock.
  - A poor scheduling policy can cause starvation
    - If a high-priority process always prevents a low-priority process from running on the CPU.
  - Synchronization can also cause starvation
    - One thread always beats another when acquiring a lock.
    - Constant supply of readers always blocks out writers.
CPU Scheduling (5)

- **Scheduling Goals**
  - **All systems**
    - No starvation
    - Fairness: giving each process a fair share of the CPU
    - 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 system
First-Come, First-Served (FCFS)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

Process arrival order: $P_1, P_2, P_3$

- The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
First-Come, First-Served (FCFS)

Suppose that the processes arrive in the order: P2, P3, P1
- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: \((6 + 0 + 3)/3 = 3\)
- Much better than previous case
- Convoy effect
  - Short process behind long process
Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

- SJF is optimal
  - Gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
    - Could ask the user or estimate

- SJF may starve long processes
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- **SJF scheduling chart**

- **Average waiting time** = \( \frac{3 + 16 + 9 + 0}{4} = 7 \)
Estimating Next CPU Burst

- **Exponential averaging**
  - Predicting the length of next CPU burst
  - Using the length of previous CPU bursts

1. \( t_n \) = actual length of \( n^{th} \) CPU burst
2. \( \tau_{n+1} \) = predicted value for the next CPU burst
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define: \[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \]
   \[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} \]
   \[ + \ldots \]
   \[ + (1 - \alpha)\alpha t_{n-j} \]
   \[ + \ldots \]
   \[ + (1 - \alpha)^n \tau_0 \]

- **Commonly, \( \alpha \) set to \( \frac{1}{2} \)
Prediction of Next CPU Burst

CPU burst \( (t_i) \)  6  4  6  4  13  13  13  ...  

"guess" \( (\tau_i) \)  10  8  6  6  5  9  11  12  ...
Shortest Remaining Time First (SRTF)

- Preemptive version of SJF.
- If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>
Round Robin (RR)

- Ready Q is treated as a circular FIFO Q.
- Each job is given a time slice (or time quantum).
  - Usually 10-100 ms.
- Great for timesharing
  - No starvation
  - Typically, higher average turnaround time than SJF, but better response time.
- Preemptive
- What do you set the quantum to be?
  - A rule of thumb: 80% of the CPU bursts should be shorter than the time quantum.
- Treats all jobs equally
Example of RR (1)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typically
- Higher average turnaround than SJF
- Better response time
**Example of RR (2)**

- **Time quantum and context switch time**

  - **Process time = 10**
  
  - **Quantum**
    - 12
    - 6
    - 1
  
  - **Context switches**
    - 0
    - 1
    - 9

---

**SDE5007: Special Topics on IC Design II | Spring 2016 | Jinkyu Jeong (jinkyu@skku.edu)**
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem
  - Starvation – low priority processes may never execute
- Solution
  - Aging – as time progresses increase the priority of the process
Priority Inversion Problem

- A situation where a higher-priority job is unable to run because a lower-priority job is holding a resource it needs, such as a lock.

- What really happened on Mars?
Solutions to Priority Inversion

- **Priority inheritance protocol (PIP)**
  - The higher-priority job can *donate* its priority to the lower-priority job holding the resource it requires.

- **Priority ceiling protocol (PCP)**
  - The priority of the low-priority thread is *raised immediately* when it gets the resource.
  - The priority ceiling value must be predetermined.
Multilevel Queue

- Ready queue is partitioned into separate queues
  - **Foreground** (interactive)
  - **Background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - Foreground – RR
  - Background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling
    - i.e.) serve all from foreground then from background
    - Possibility of starvation.
  - Time slice
    - Each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - i.e.) 80% to foreground in RR and 20% to background in FCFS
Multilevel Queue Scheduling

- highest priority
  - system processes
  - interactive processes
  - interactive editing processes
  - batch processes
  - student processes
- lowest priority
Multilevel Feedback Queue

- Multilevel queue scheduling, which allows a job to move between the various queues.
- Queues have priorities.
  - Batch, interactive, system, CPU-bound, I/O-bound, ...
- When a process uses too much CPU time, move to a lower-priority queue.
  - Leaves I/O-bound and interactive processes in the higher-priority queues.
- When a process waits too long in a lower priority queue, move to a higher-priority queue.
  - Prevents starvation.
UNIX Scheduler (1)

- **Characteristics**
  - Preemptive
  - Priority-based
    - The process with the highest priority always runs.
    - 3 – 4 classes spanning ~170 priority levels (Solaris 2)
  - Time-shared
    - Based on timeslice (or quantum)
  - MLFQ (Multi-Level Feedback Queue)
    - Priority scheduling across queues, RR within a queue.
    - Processes dynamically change priority.
UNIX Scheduler (2)

- **General principles**
  - Favor I/O-bound processes over CPU-bound processes
    - I/O-bound processes typically run using short CPU bursts.
    - Provide good interactive response; don’t want editor to wait until CPU hog finishes quantum.
    - CPU-bound processes should not be severely affected.
  - No starvation
    - Use aging
Linux Scheduling in 2.6.23+

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Real-time classes: SCHED_FIFO, SCHED_RR
  - Default (fair-share) class
    - Tasks share CPU time proportionally
- **Quantum calculated based on nice value from -20 to +19**
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if number of active tasks increases
- **CFS scheduler maintains per task virtual runtime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- **Scheduler picks next task with lowest virtual runtime**
  - Task who has had the lowest CPU time
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of \( v\text{runtime} \). This tree is shown below:

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of \( v\text{runtime} \)) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require \( O(\log N) \) operations (where \( N \) is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable \text{rb}	extunderscore leftmost, and thus determining which task to run next requires only retrieving the cached value.