Operating Systems

Processes and Threads - Part 2

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

http://www.mzahran.com

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—*THE SCIENCES OF THE ARTIFICIAL,*
Herbert Simon

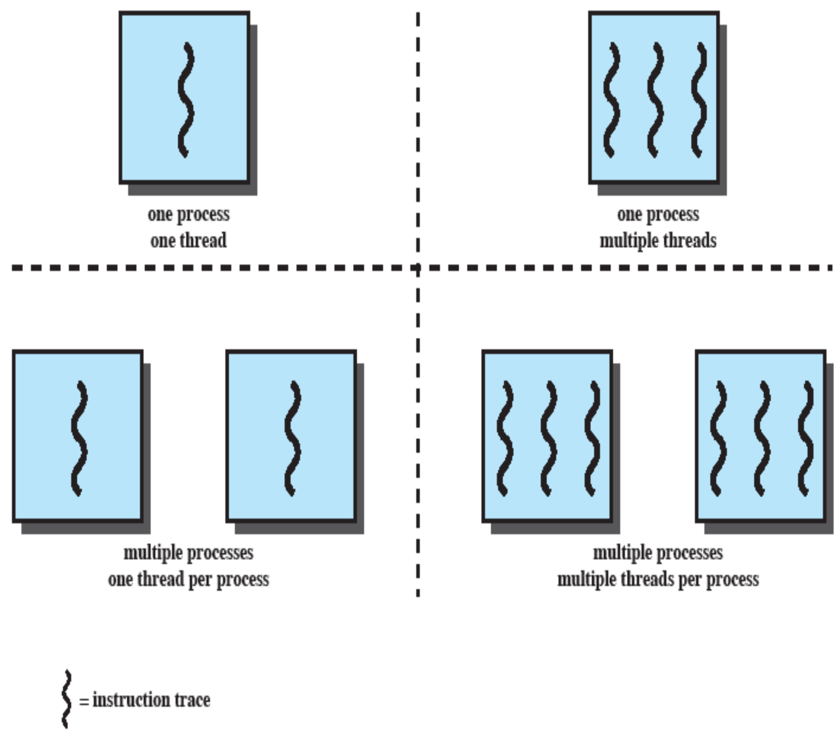
Processes Vs Threads

- A path of execution is referred to as a ***thread***
- The unit of resource ownership is referred to as a ***process***
- ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Processes Vs Threads

- Process is the unit for resource allocation and a unit of protection.
- Process has its own address space.
- A thread has:
  - an execution state (Running, Ready, etc.)
  - saved thread context when not running
  - an execution stack
  - some per-thread static storage for local variables
  - access to the memory and resources of its process (all threads of a process share this)

A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach … Example: MS-DOS



A Java run-time environment is an example of a system of one process with multiple threads.

Benefits of Threads

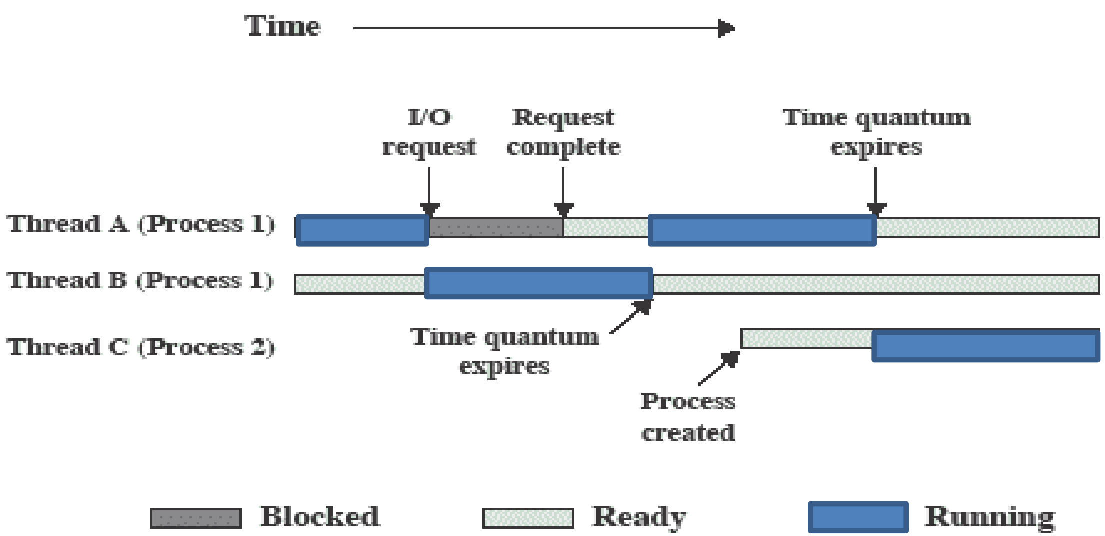
Takes less time to create a new thread than a process

Less time to terminate a thread than a process

Switching between two threads takes less time than switching between processes

Threads enhance efficiency in communication between programs

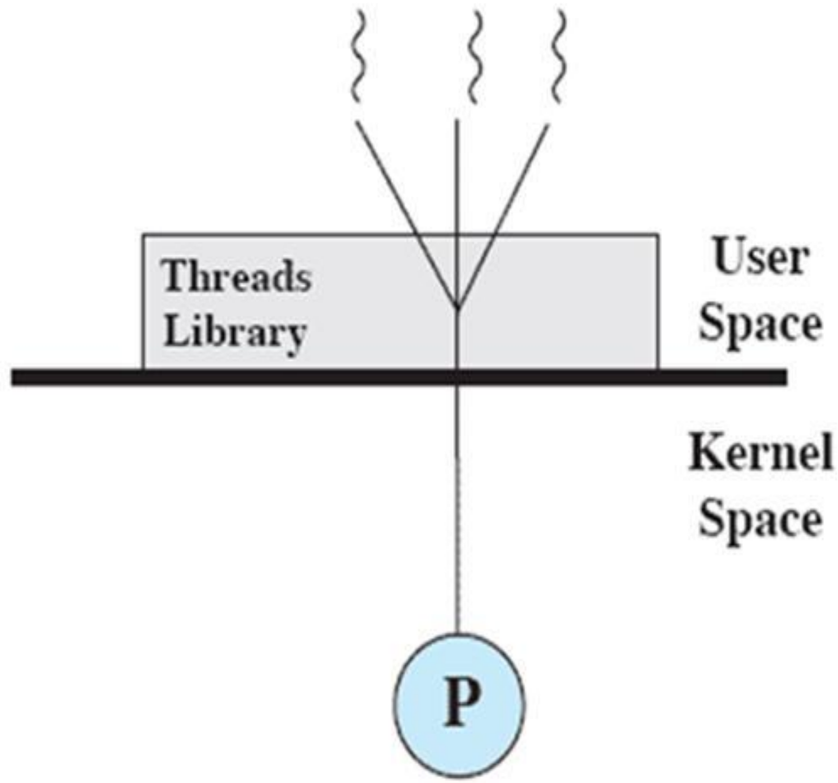
Multithreading on Uniprocessor System



User-level Threads
vs
Kernel-level Threads

User-Lever Threads (ULT)

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
- The kernel assigns the whole process as a single unit.



User-Level Threads (ULTs)

Advantages

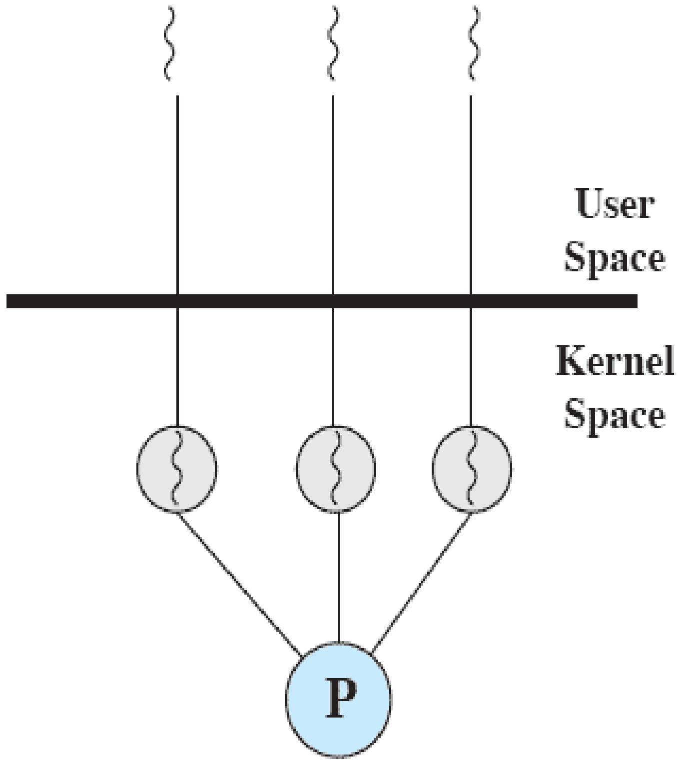
- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.

Disadvantages

- A system-call by one thread can block all threads of that process.
- In pure ULT, multithreading cannot take advantage of multiprocessing/multicore

Kernel-Level Threads (KLTs)

- Thread management is done by the kernel.
- No thread management is done by the application.
- Windows OS is an example of this approach.



Kernel-Level Threads (KLTs)

Advantages

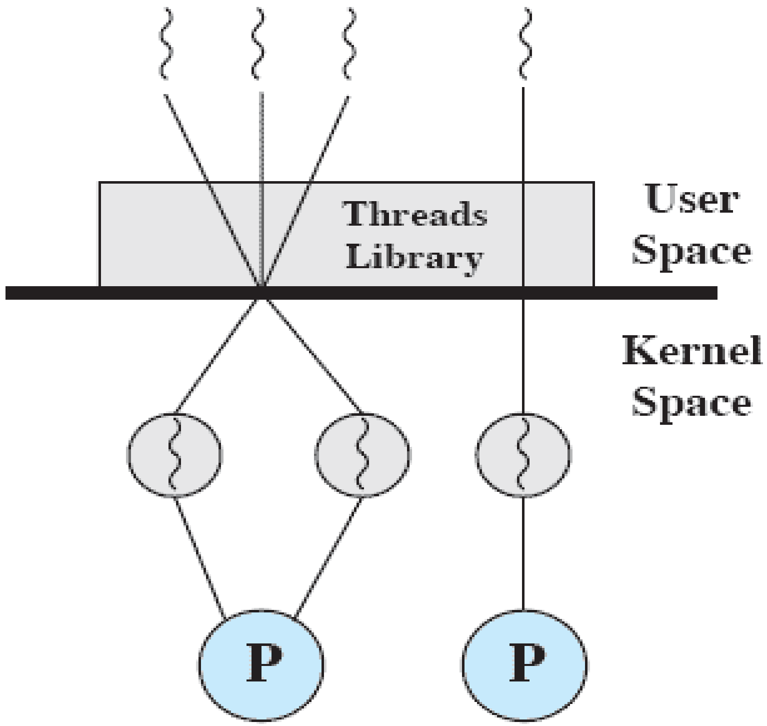
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors/cores.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.

Disadvantages

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example.



Threads and Processes Relationship

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

APIs for dealing with Processes in C

Don't forget to include the following header files:

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

Basic Syscalls for Managing Processes

- `fork` spawns new process
  - Called once, returns twice
- `exit` terminates own process
  - Puts it into "zombie" status until its parent reaps
- `wait` and `waitpid` wait for and reap terminated children
- `execve` runs new program in existing process
  - Called once, never returns

`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
- Fork is called *once* but returns *twice*

Return 0 to the child process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent: child pid is %d\n", pid);
}
```

Return child's pid to the parent

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from parent
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from child
```

Fork Example #1

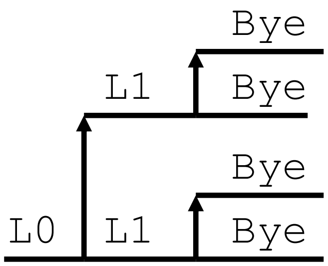
- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state, but each has private copy of memory
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- Both parent and child can continue forking

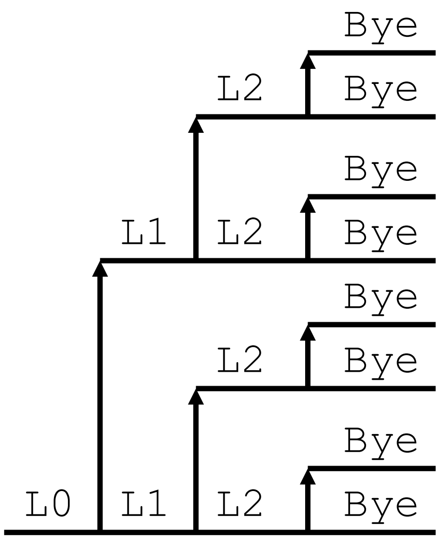
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Both parent and child can continue forking

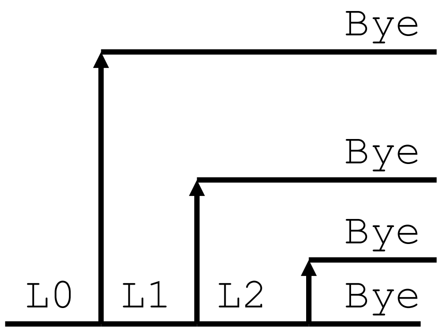
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Both parent and child can continue forking

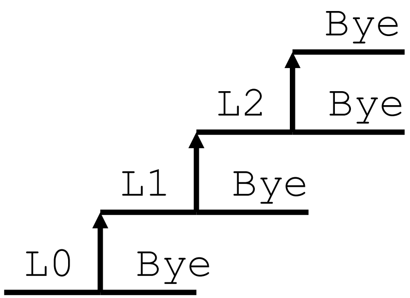
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



`exit`: Ending a process

- `void exit(int status)`
  - exits a process
    - Normally return with status 0
  - `atexit(function_name)` make function_name execute upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies!

- Idea
  - When process terminates , still consumes system resources (i.e. an entry in process table)
    - Why? So that parents can learn of children's exit status
  - Called a "zombie"
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - OS discards process
- What if parent doesn't reap?
  - If parent has terminated, then child will be reaped by `init` process (the great-great-…-grandparent of all user-level processes)

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1) ; /* Infinite loop */
    }
}
```

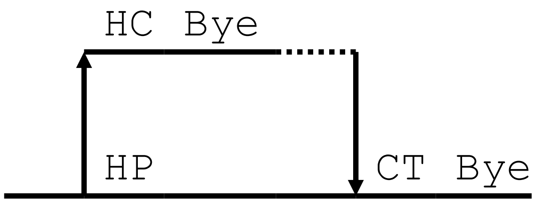
```
linux> ./forks7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`

`wait`: Synchronizing with Children

- `int wait(int *child_status)`
  - Blocks until some child exits, return value is the **`pid`** of terminated child
  - If multiple children completed, will take in arbitrary order (use **`waitpid`** to wait for a specific child)

```
void fork8() {
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit(0);
}
```



This is how child process is reaped by parent process.

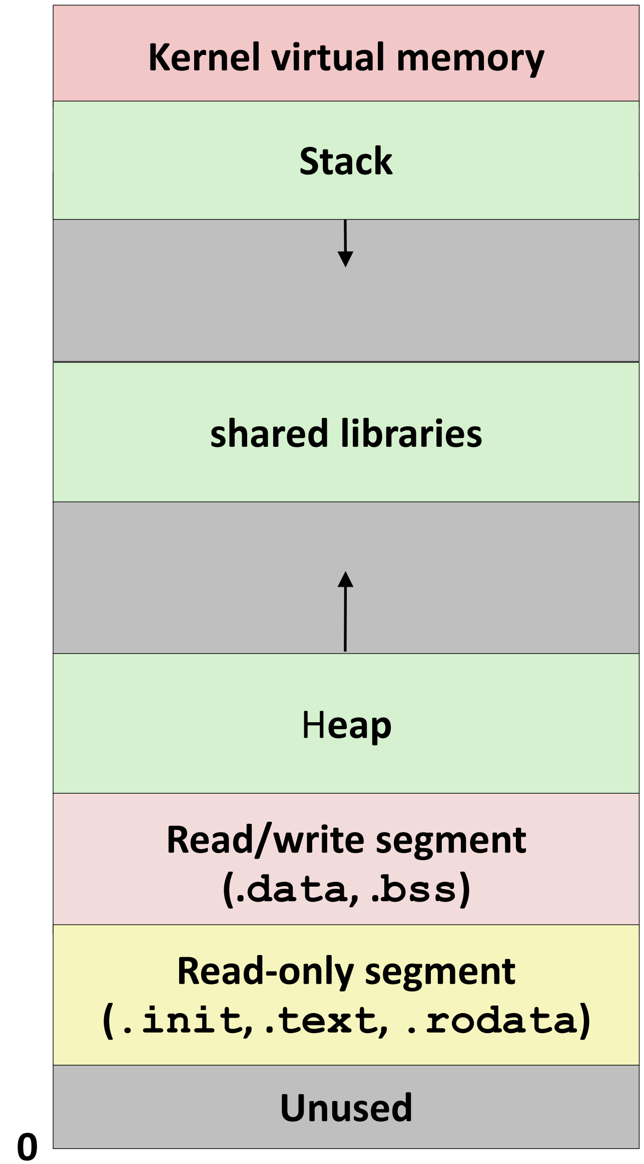
execve

- **int execve(char *fname, char *argv[],** `char *envp[])`
  - Executes program named by `fname`

```
if ((pid = fork()) == 0) { /* Child runs user job */
     if (execve(argv[0], argv, environ) < 0) {
          printf("%s: Command not found.\n", argv[0]);
          exit(0);
     }
}
```

`execve`: Load a new program image

- `execve` causes OS to overwrite code, data, and stack of process
  - keeps pid and open files

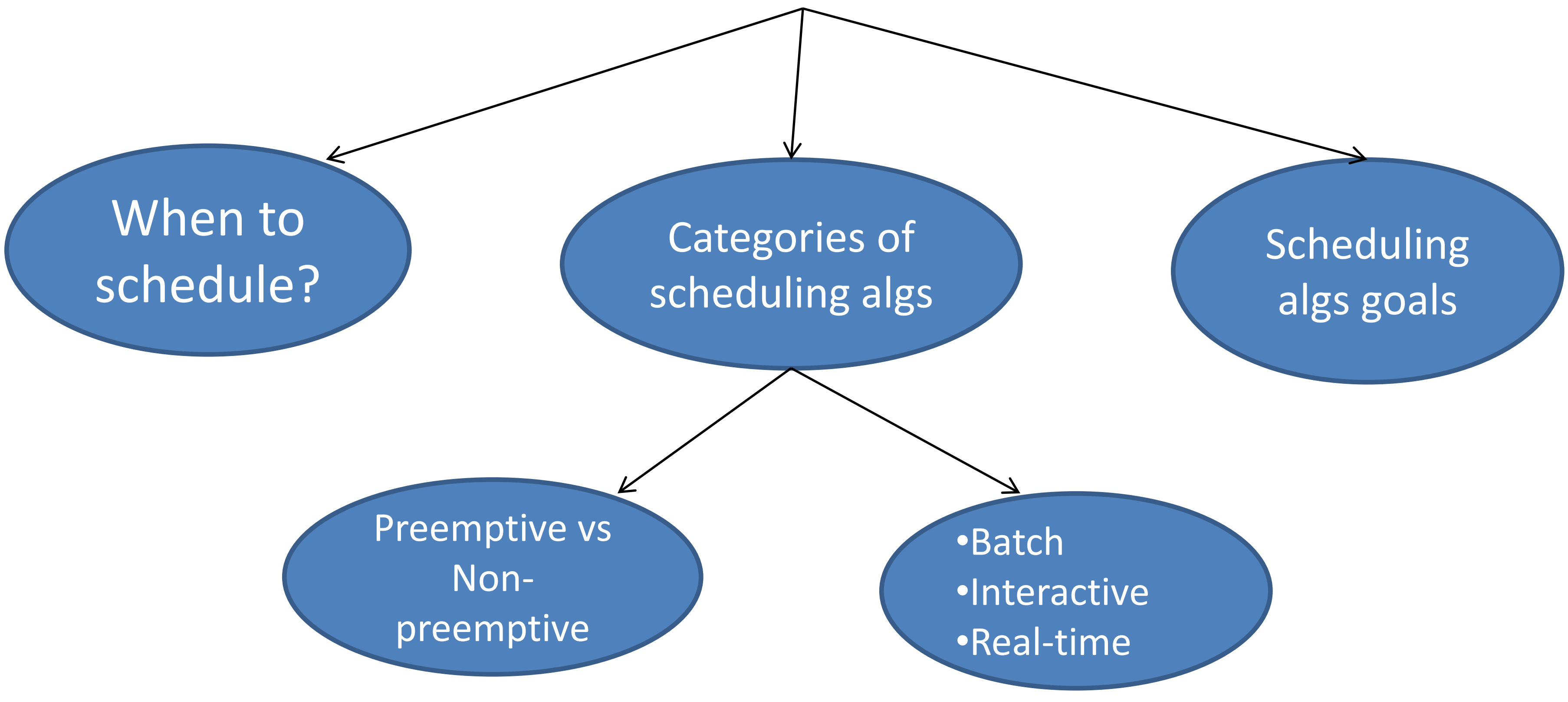


Scheduling

Given a group of ready processes, which process to run next?

Scheduling

Given a group of ready processes, which process to run next?



Definition

- Preemptive scheduling: A process can be interrupted, and another process scheduled to execut2.
- Non-preemptive scheduling: The current running process mu finish/exit first before another process is schedule to execute.

When to Schedule?

- When a process is created
- When a process exits
- When a process blocks
- When an I/O interrupt occurs

Categories of Scheduling Algorithms

- Batch
  - No users impatiently waiting
  - mostly non-preemptive
- Interactive
  - preemption is essential
- Real-time
  - deadlines

Scheduling Algorithms Goals

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

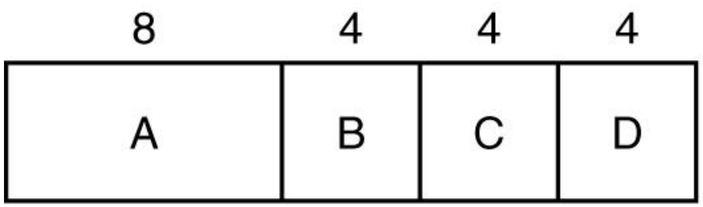
Meeting deadlines - avoid losing data

Scheduling in Batch Systems: First-Come First-Served

- Non-preemptive
- Processes ordered as queue
- A new process added to the end of the queue
- A blocked process that becomes ready added to the end of the queue
- Main disadv: Can hurt I/O bound processes

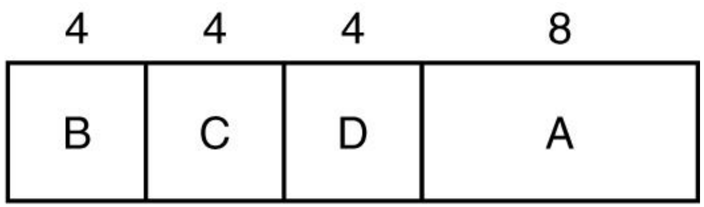
Scheduling in Batch Systems: Shortest Job First

- Non-preemtive
- Assumes runtime is known in advance
- Is only optimal when all the jobs are available simultaneously



(a)

Run in original order



(b)

Run in shortest job first

Scheduling in Batch Systems: Shortest Remaining Time Next

- Preemptive
- Scheduler always chooses the process whose remaining time is the shortest.
- Runtime must be known in advance.

Scheduling in Interactive Systems: Round-Robin

- Each process is assigned a time interval: quantum (or time slice)
- After this quantum, the CPU is given to another process
- What is the length of this quantum?
  - too short -> too many context switches -> lower CPU efficiency
  - too long -> poor response to short interactive

Scheduling in Interactive Systems: Priority Scheduling

- Each process is assigned a priority.
- Ready process with the highest priority is allowed to run.
- Priorities are assigned statically or dynamically.
- Must not allow a process to run forever
  - Can decrease the priority of the currently running process.
  - Use time quantum for each process.

Scheduling in Real-Time

- Process must respond to an event within a deadline.
- Hard real-time vs soft real-time
  - Hard: Result/Response becomes incorrect if you miss the deadline. Used in critical applications like medical, air-traffic control, … .
  - Soft: If deadline is missed, system is still correct but with degraded performance. Example: computer games.
- Periodic vs aperiodic events
- Processes must be schedulable
- Scheduling algorithms can be static or dynamic

Thread Scheduling

- Similar algorithms as processes.
- Scheduling can be done at:
  - user space: in user-level threads
  - kernel space: in kernel-level threads

Conclusion

- Threads and processes are crucial concepts in OS design.
- As OS designer, you must make decision regarding: process table, threading, scheduling, etc.
- We have covered more stuff than the book so you may find information here more than the book (especially in mutual exclusion part).