

Last Class: Processes

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
 - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a *context switch*
- Processes communicate either with message passing or shared memory



Example Unix Program: Fork

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

main() {
    int parentID = getpid();    /* ID of this process */
    char prgname[1024];
    gets(prgname); /* read the name of program we want to start */
    int cid = fork();
    if(cid == 0) { /* I'm the child process */
        execlp( prgname, prgname, 0); /* Load the program */
        /* If the program named prgname can be started, we never get
        to this line, because the child program is replaced by prgname */
        printf("I didn't find program %s\n", prgname);
    } else { /* I'm the parent process */
        sleep (1); /* Give my child time to start. */
        waitpid(cid, 0, 0); /* Wait for my child to terminate. */
        printf("Program %s finished\n", prgname);
    } }
}
```



Example Unix Program: Explanation

fork() forks a new child process that is a copy of the parent.

execlp() replaces the program of the current process with the named program.

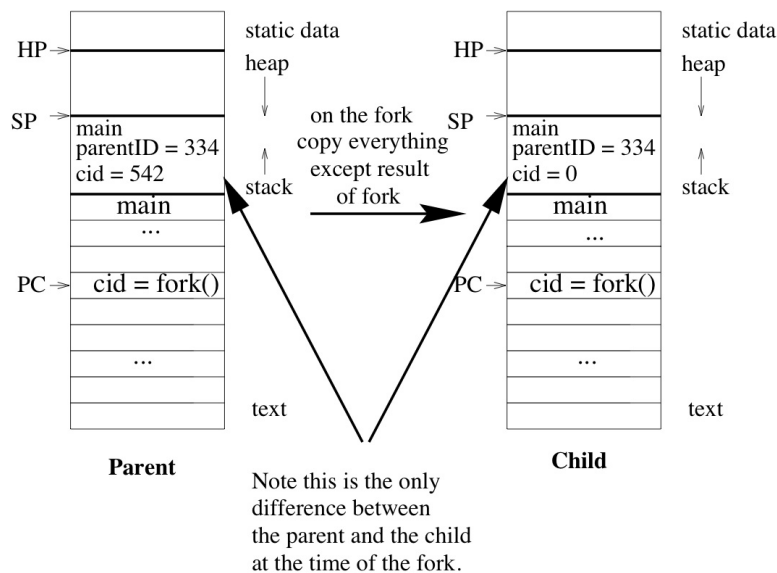
sleep() suspends execution for at least the specified time.

waitpid() waits for the named process to finish execution.

gets() reads a line from a file.



What is happening on the Fork



Process Termination

- On process termination, the OS reclaims all resources assigned to the process.
- In Unix
 - a process can terminate itself using the *exit* system call.
 - a process can terminate a child using the *kill* system



Example Unix Program: Process Termination

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
main() {
    int parentID = getpid();    /* ID of this process */
    int cid = fork();
    if(cid == 0) { /* I'm the child process */
        sleep (5);    /* I'll exit myself after 5 seconds. */
        printf ( "Quitting child\n" );
        exit (0);
        printf ( "Error! After exit call.!" ); /* should never get here
        */
    } else { /* I'm the parent process */
        printf ( "Type any character to kill the child.\n" );
        char answer[10];
        gets (answer);
        if ( !kill(cid, SIGKILL) ) {
            printf("Killed the child.\n");
        }
    }
}
```



Cooperating Processes

- Any two process are either independent or cooperating
 - Cooperating processes work with each other to accomplish a single task.
 - Cooperating processes can
 - improve performance by overlapping activities or performing work in parallel,
 - enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program, and
 - easily share information between tasks.
- Distributed and parallel processing is the wave of the future. To program these machines, we must cooperate and coordinate between separate processes.



Cooperating Processes: Producers and Consumers

```
n = 100 //max outstanding items
in = 0
out = 0
```

producer

```
repeat forever{
  ...
  nextp = produce item
  while in+1 mod n = out
    do no-opt
  buffer[in] = nextp
  in = in+1 mod n
}
```

consumer

```
repeat forever{
  //Make sure buffer not empty
  while in = out do no-opt
  nextc = buffer[out]
  out = out+1 mod n
  ...
  consume nextc
}
```

- Producers and consumers can communicate using *message passing* or *shared memory*



Communication using Message Passing

```
main()
...
if (fork() != 0) producerSR;
else consumerSR;
end

producerSR
repeat
...
produce item nextp
...
send(nextp, consumer)

consumerSR
repeat
receive(nextc, producer)
...
consume item nextc
...
```



Message Passing

- Distributed systems typically communicate using message passing
- Each process needs to be able to name the other process.
- The consumer is assumed to have an infinite buffer size.
- A bounded buffer would require the tests in the previous slide, and communication of the **in** and **out** variables (**in** from producer to consumer, **out** from consumer to producer).
- OS keeps track of messages (copies them, notifies receiving process, etc.).

→ How would you use message passing to implement a single producer and multiple consumers?



Communication using Shared Memory

- Establish a mapping between the process's address space to a named memory object that may be shared across processes
- The `mmap(...)` systems call performs this function.
- Fork processes that need to share the data structure.



Shared Memory Example

main()

```
...
mmap(..., in, out, PROT_WRITE, PROT_SHARED, ...);
in = 0;
out = 0;
if (fork != 0) produce();
else consumer();
end
```

producer

repeat

```
...
produce item nextp
...
while in+1 mod n = out do no-opt
buffer[in] = nextp
in = in+1 mod n
```

consumer

repeat

```
while in = out do no-op
nextc = buffer[out]
out = out+1 mod n
...
consume item nextc
...
```



Today: Threads

- What are threads?
- Where should we implement threads? In the kernel? In a user level threads package?
- How should we schedule threads (or processes) onto the CPU?

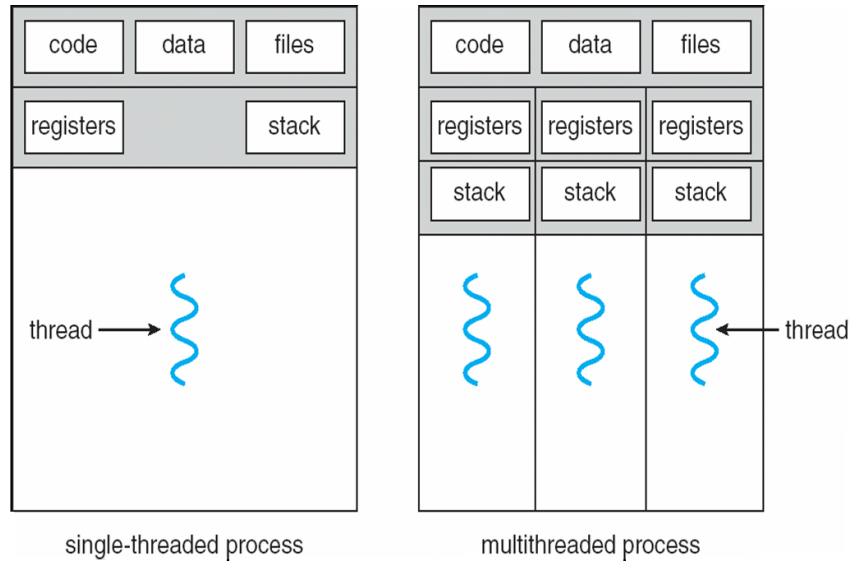


Processes versus Threads

- A **process** defines the address space, text, resources, etc.,
- A **thread** defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the *thread of control* information from the process
- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
 - The address space of a process is shared among all its threads
 - No system calls are required to cooperate among threads
 - Simpler than message passing and shared-memory

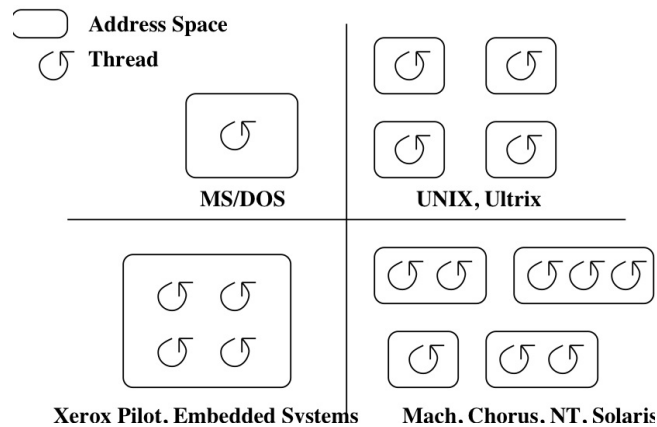


Single and Multithreaded Processes



Classifying Threaded Systems

Operating Systems can support one or many address spaces, and one or many threads per address space.



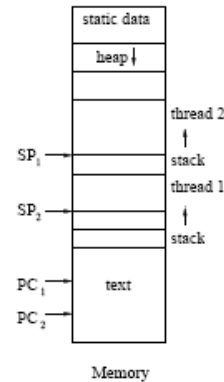
Example Threaded Program

```
main()
  global in, out, n, buffer[n];
  in = 0; out = 0;
  fork_thread (producer());
  fork_thread (consumer());
end

producer
  repeat
    nextp = produced item
    while in+1 mod n = out do no-op
    buffer[in] = nextp; in = (in+1) mod n
  end

consumer
  repeat
    while in = out do no-op
    nextc = buffer[out]; out = (out+1) mod n
    consume item nextc
  end
```

One possible
memory layout:



- Forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code).



Kernel Threads

- A **kernel thread**, also known as a **lightweight process**, is a thread that the operating system knows about.
 - Switching between kernel threads of the same process requires a small context switch.
 - The values of registers, program counter, and stack pointer must be changed.
 - Memory management information does not need to be changed since the threads share an address space.
 - The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.
- Switching between kernel threads is slightly faster than switching between processes.

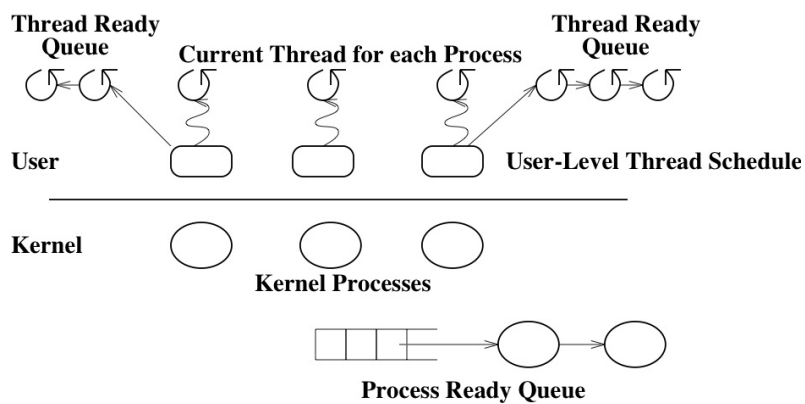


User-Level Threads

- A **user-level thread** is a thread that the OS does *not* know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a *thread library* to manage threads (create and delete them, synchronize them, and schedule them).



User-Level Threads



User-Level Threads: Advantages

- There is no context switch involved when switching threads.
 - User-level thread scheduling is more flexible
 - A user-level code can define a problem dependent thread scheduling policy.
 - Each process might use a different scheduling algorithm for its own threads.
 - A thread can voluntarily give up the processor by telling the scheduler it will *yield* to other threads.
 - User-level threads do not require system calls to create them or context switches to move between them
- User-level threads are typically much faster than kernel threads

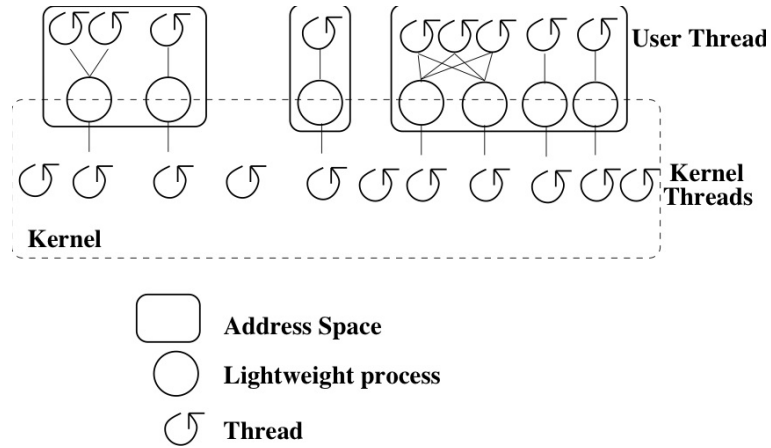


User-Level Threads: Disadvantages

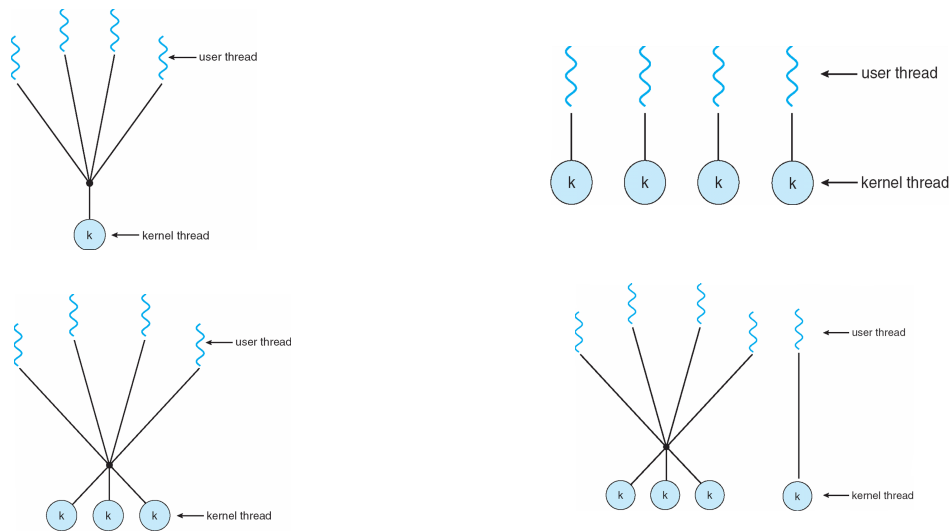
- Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
 - It might run a process that only has idle threads.
 - If a user-level thread is waiting for I/O, the entire process will wait.
 - Solving this problem requires communication between the kernel and the user-level thread manager.
- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads.
- For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.



Example: Kernel and User-Level Threads in Solaris



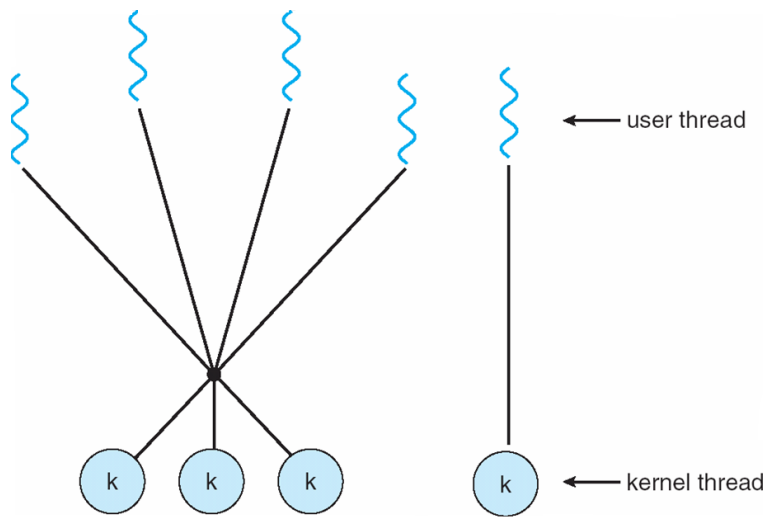
Threading Models



- Many-to-one, one-to-one, many-to-many and two-level



Two-level Model



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS



Pthreads

- May be provided either as user-level or kernel-level
- 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 (Solaris, Linux, Mac OS X)

- WIN32 Threads: Similar to Posix, but for Windows



Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface



Examples

Pthreads:

```
pthread_attr_init(&attr); /* set default attributes */  
pthread_create(&tid, &attr, sum, &param);
```

Win32 threads

```
ThreadHandle = CreateThread(NULL, 0, Sum, &Param, 0, &ThreadID);
```

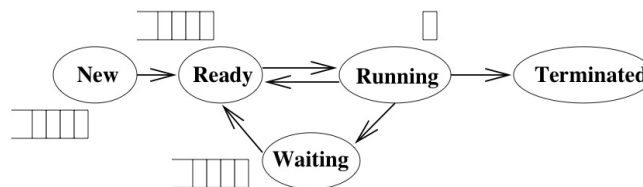
Java Threads:

```
Sum sumObject = new Sum();  
Thread t = new Thread(new Summation(param, SumObject));  
t.start(); // start the thread
```



Scheduling Processes

- **Multiprogramming:** running more than one process at a time enables the OS to increase system utilization and throughput by overlapping I/O and CPU activities.
- Process Execution State



- All of the processes that the OS is currently managing reside in one and only one of these state queues.



Scheduling Processes

- **Long Term Scheduling:** How does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory?
- **Short Term Scheduling:** How does (or should) the OS select a process from the ready queue to execute?
 - Policy Goals
 - Policy Options
 - Implementation considerations



Short Term Scheduling

- The kernel runs the scheduler at least when
 1. a process switches from running to waiting,
 2. an interrupt occurs, or
 3. a process is created or terminated.
- **Non-preemptive system:** the scheduler must wait for one of these events
- **Preemptive system:** the scheduler can interrupt a running process



Criteria for Comparing Scheduling Algorithms

- **CPU Utilization** The percentage of time that the CPU is busy.
- **Throughput** The number of processes completing in a unit of time.
- **Turnaround time** The length of time it takes to run a process from initialization to termination, including all the waiting time.
- **Waiting time** The total amount of time that a process is in the ready queue.
- **Response time** The time between when a process is ready to run and its next I/O request.



Scheduling Policies

Ideally, choose a CPU scheduler that optimizes all criteria simultaneously (utilization, throughput,..), but this is not generally possible

Instead, choose a scheduling algorithm based on its ability to satisfy a policy

- Minimize average response time - provide output to the user as quickly as possible and process their input as soon as it is received.
- Minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance.
- Maximize throughput - two components
 - minimize overhead (OS overhead, context switching)
 - efficient use of system resources (CPU, I/O devices)
- Minimize waiting time - give each process the same amount of time on the processor. This might actually increase average response time.



Scheduling Policies

Simplifying Assumptions

- One process per user
- One thread per process
- Processes are independent

Researchers developed these algorithms in the 70's when these assumptions were more realistic, and it is still an open problem how to relax these assumptions.



Scheduling Algorithms: A Snapshot

FCFS: First Come, First Served

Round Robin: Use a time slice and preemption to alternate jobs.

SJF: Shortest Job First

Multilevel Feedback Queues: Round robin on each priority queue.

Lottery Scheduling: Jobs get tickets and scheduler randomly picks winning ticket.



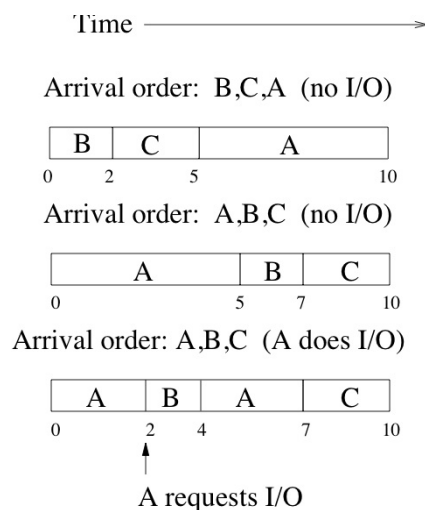
Scheduling Policies

FCFS: First-Come-First-Served (or FIFO: First-In-First-Out)

- The scheduler executes jobs to completion in arrival order.
- In early FCFS schedulers, the job did not relinquish the CPU even when it was doing I/O.
- We will assume a FCFS scheduler that runs when processes are blocked on I/O, but that is non-preemptive, i.e., the job keeps the CPU until it blocks (say on an I/O device).



FCFS Scheduling Policy: Example



- If processes arrive 1 time unit apart, what is the average wait time in these three cases?



FCFS: Advantages and Disadvantages

Advantage: simple

Disadvantages:

- average wait time is highly variable as short jobs may wait behind long jobs.
- may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle



Summary

- Thread: a single execution stream within a process
- Switching between user-level threads is faster than between kernel threads since a context switch is not required.
- User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.
- Many scheduling algorithms exist. Selecting an algorithm is a policy decision and should be based on characteristics of processes being run and goals of operating system (minimize response time, maximize throughput, ...).

