From last week:

In a threaded environment, instead of there being several singly threaded processes, we have one process consisting of multiple threads. In other words, each user process can have several threads of execution — different parts of your code can be executing simultaneously. The collection of threads form a single process: each thread in the process sees the same virtual address space, files, etc.

This last fact has interesting implications. For example, when one thread opens a file, it is immediately possible for all other threads to access the file. Similarly, if one thread alters the value of a global variable, all other threads instantly see the new value.

When writing multithreaded applications, you must take the initiative to protect shared resources. For example, consider the code:

```
i = i + 1
```

This statement is well defined for a single thread of execution: get `i`, add 1, and put the result back. For multiple threads of execution, the statement is **not** well defined!

If 2 or more threads attempt to perform the operation:

```
i = i + 1
```

simultaneously, the result is **indeterminate**.

The variable `i` will get incremented anywhere from one to N times, where N is the number of threads that execute this statement.

In a multithreaded environment, it is necessary for the programmer to ensure that access to shared resources is managed in a well defined fashion.

In order to deal with the "`i=i+1` problem", it is necessary to invent some kind of locking mechanism that serializes access to the variable `i`.

Pthreads provides an object called a **mutex** (short for "mutual exclusion") to facilitate resource locking. Once you create a mutex, there are two things you can do with it:

lock it
unlock it

Mutexes have the property that they can only be locked by one thread at a time. If two threads attempt to lock the same mutex, only one of them succeeds. The other one blocks until the first thread releases the mutex.

In general, you use mutexes to protect some piece of shared data. In order to be useful, the mutex itself must be global so that all threads can see it (this is true of all pthreads synchronization objects).


```
int i;                        /* i is shared */
pthread_mutex_t ilock;        /* controls access to i */

void func()
{


pthread_mutex_lock(&ilock);   /* lock i */
something = i++;              /* access i */
pthread_mutex_unlock(&ilock);/* unlock i */


}
```

It is up to the programmer to determine exactly what the mutex protects. Here the mutex `ilock` is used to protect `i` (it serializes the access to `i`).

All global data that can be read and written by simultaneously executing threads must be protected by a mutex.

In order to minimize ill effects, it is important to "get in and out" as quickly as possible. For example, if a thread needs to read some global data, consider copying the data into the thread's private storage instead of keeping a mutex locked for a long period of time.

Or, if you need to update a piece of global data, calculate (if possible) the new value with the mutex unlocked – lock it only when you dump the new value in.

Why Use Threads?

Why should you use threads? Well, maybe you shouldn't. Developing a threaded program can take longer than a nonthreaded program. If you don't need the benefits of threads or don't intend to run your code many times, it's probably not worth the trouble. On the other hand, threading your code has several payoffs:

Speed: Communication between threads is very fast. If your algorithm is bound by communication speed, threads will put a smile back on your face.

I/O throughput: In a traditional UNIX process, a blocked I/O request stops the whole process dead in its tracks. Things are much nicer in POSIX threads: a blocked I/O request only puts the calling thread to sleep – the other threads go about their business.

Portability: POSIX threads are fairly portable. They can be emulated on uniprocessor systems at a slight (or sometimes no) penalty in speed. On shared memory multiprocessor machines, there is often kernel support for threads; this means that your application can take full advantage of all available CPUs and achieve (near) maximum performance.

Practice Exercises:

From the textbook: 5.1–6

5.1 Provide two programming examples of multithreading that improve performance over a single-threaded solution:

–A Web server that services each request in a separate thread

–A parallelized application such as matrix multiplication where different parts of the matrix may be worked in parallel.

–An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

5.3 What are two differences between user-level threads and kernel-level threads?

–User-lever threads are unknown by the kernel, whereas the kernel is aware of kernel threads.

–User threads are scheduled by the thread library (such as Pthreads for C) and the kernel schedules kernel threads.

–Kernel threads need not be associated with a process whereas every user thread belongs to a process.

5.4 Describe the actions taken by a kernel to context switch between kernel-level threads.

Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

5.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environmental variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread invloves allocating a small data structure to hold a register set, stack, and priority.

Developing New Operating Systems

(from a COMPUTERWORLD article by Gary H. Anthes)

Research and development work on operating systems, broadly defined, is going on in many quarters. Here are some projects:

The Legion Project:

Researchers at the University of Virginia are developing Legion, a highly flexible, wide-area operating system designed to build a virtual computer from millions of distributed hosts and trillions of objects while presenting to the user the image of a single computer. Legion applies many of the object-based interoperability principles in the Common Object Request Broker Architecture. It finds and schedules resources and handles security issues among disparate operating systems and objects written in different languages, freeing users from the need to negotiate with outside systems and administrators.

The Globus Project:

This is a university/government research and development project focused on applying grid concepts to scientific and engineering computing. The grid refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases and scientific instruments owned and managed by multiple organizations. Grid applications often involve large amounts of data and/or computing and often require secure resource sharing across organizational boundaries.

Globus research areas include resource management, data management and access, application development environments, information services and security. The project has developed the Globus Toolkit, a set of services and software libraries to support Grids and Grid applications. The Globus Toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection and portability.

The Endeavour project:

Researchers involved in the Endeavour project at the University of California, Berkeley, envision an "information utility" that adapts on the fly to the supply and demand for computing resources anywhere in the world. "Fluid software" will choose where to execute and where to access storage. It will requisition resources as needed and negotiate interoperation agreements, ensuring always-available processing. Endeavour is a collection of projects. One, called OceanStore, will provide "nomadic data access – serverless, homeless and freely flowing through the network" – that will make the storage available to a user essentially infinite, according to researchers.

The WebOS project:

The goal of this research effort at Duke University in Durham, N.C., is to provide a common set of operating system services to wide-area applications, including mechanisms for resource discovery, a global name space, remote process execution, resource management, authentication and security. WebOS provides basic operating system services needed to build applications that are geographically distributed, highly available, incrementally scalable and dynamically reconfiguring.

The Globe Project:

Researchers involved in The Globe Project at Vrije University in Amsterdam are working on construction of large-scale wide-area distributed systems enabled by distributed shared objects. In the Globe model, the universe consists of a vast number of shared objects, each with some associated methods. Users of an object may invoke an object's methods. A method can return the contents of a Web page, but it can also accept an e-mail or news message, look up a name in a worldwide distributed database, access a file and perform other actions. Objects are distributed, with active copies on multiple machines at the same time, and they use peer-to-peer communication.

MIT Project Oxygen:

Sponsored by the U.S. Department of Defense, this project at MIT is intended to bring computation and communication to users through natural spoken and visual interfaces, in order to make it easy for users to collaborate, access knowledge and automate repetitive tasks. The Oxygen system rests on an infrastructure of mobile and stationary devices connected by a self-configuring network. Researchers say the Oxygen system must be:

- Pervasive: It must be everywhere, with every portal reaching into the same information base.
- Embedded: It must live in its users' world, sensing and affecting it.
- Nomadic: Its users and computations must be free to move around according to their needs.
- Eternal: It must never shut down or reboot; components may come and go in response to demand, errors and upgrades, but Oxygen as a whole must be nonstop.

Correction to Question 2b on Test 1

Count the number of faults that OPT incurs on $\sigma_{k,i}$. Suppose, for example, that $k = 2, i = 4$. Then:

$$\sigma_{2,4} = 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3$$

My solution said that the number of faults is OPT$(\sigma_{k,i}) = k + i = 6$. However:

$$\begin{array}{|c|c|c|}1 & 1 & 1\\ & 2 & 3\\ \text{x} & \text{x} & \text{x}\end{array}, \begin{array}{|c|c|c|}1 & 2 & 2\\ 3 & 3 & 3\\ & \text{x} & \end{array}, \begin{array}{|c|c|c|}2 & 2 & 3\\ 1 & 1 & 1\\ \text{x} & & \text{x}\end{array}, \begin{array}{|c|c|c|}3 & 3 & 3\\ 1 & 2 & 2\\ & \text{x} & \end{array}$$

So, the number of faults is 7, not 6. It turns out that every k segments (AFTER the first segment) there is one extra fault. Thus, the correct formula for OPT$(\sigma_{k,i})$ is the following:

$$k + i + \left\lfloor \frac{i-1}{k} \right\rfloor$$

(the $i - 1$ comes from counting the segments without the first segment, and then, to add up the extra faults that occur every k-th segment, we divide by k and take the floor). Try this with $\sigma_{4,9}$ to see this extra fault on another example.

Note that this is a "small" addition to the number of faults incurred by OPT. Thus, 2c is still correct (note that OPT$(\sigma_{k,i}) = k + i + \lfloor \frac{i-1}{k} \rfloor \leq 2k + i$, so now:

$$\text{FIFO}(\sigma_{k,i}) \leq \left(\frac{i(k+1)}{2k+i}\right) \text{OPT}(\sigma_{k,i})$$

so the same analysis still holds.

CPU Scheduling

CPU Scheduling is the basis of multiprogrammed OSs. By switching the CPU among processes, the OS can make the computer more productive.

The object of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. A process is executed until it must wait, typically for the completion of some I/O request. When a process has to wait, the OS gives the CPU to another process.

Scheduling is a fundamental OS function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, CPU scheduling is central to OS design.

Process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**, which is then followed by an **I/O burst**, and so on, they alternate.

The durations of these CPU bursts have been extensively measured and researched, but it is difficult to make general assertions about them. However, they can be roughly characterized as follows:

–few long CPU bursts
–many short CPU bursts
–many very short I/O bursts

Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**.

CPU scheduling decisions may take place under the following four circumstances: a process switches

1. running state $\rightarrow$ waiting state (e.g., I/O request)
2. running state $\rightarrow$ ready state (e.g., interrupt)
3. waiting state $\rightarrow$ ready state (e.g., I/O completion)
4. or a process terminates

In 1 and 4 a new process **must** be selected; when scheduling takes place only under circumstances 1 and 4, it is **nonpreemptive**. Otherwise, it is **preemptive**.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting queue.

Windows 3.1 and Apple Macintosh are nonpreemptive. Linux allows only user processes to be preempted.

The **dispatcher** is the module that gives control of the CPU to the process selected by the CPU scheduler. His work involves:

- Switching control
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, given that it is involved during every process switch. The time taken by the dispatcher to stop one process and start another is known as the **dispatch latency**.

Scheduling Criteria

- **CPU utilization:** Want to keep the CPU as busy as possible.
- **Throughput:** The number of processes completed per time unit.
- **Turnaround time:** The time interval from the submission of a process to its completion.
- **Waiting time:** It is the sum of the periods spent waiting in the ready queue.
- **Response time:** The time from the submission of a request until the first response is produced.

In most cases, we want to optimize the average measure. In some cases, we want to optimize max and min values, such as response time, so that users get good service.

Scheduling Algorithms

First-Come, First-Served (FCFS)

With this scheme, the process that requests the CPU first is allocated the CPU first.

Implementation: FIFO queue.

When a process enters the ready queue, its PCB is linked to the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue, and the running process is removed from the queue.

Problem: Average waiting time quite long. For example:

Processes (in order of arrival)	Burst Time
P_1	24
P_2	3
P_3	3

(Gantt chart)

Convoy effect: Many short processes wait for one long process.

The FCFS algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU. It is not very good for time-sharing systems, where each user needs to get a share of the CPU at regular intervals.

Shortest-Job-First Scheduling (SJF)

This algorithms associates with each process the length of its next CPU burst.

When the CPU is available, it is assigned the process that has the smallest next CPU burst. If two process have the same length next CPU burst, FCFS scheduling is used to break the tie.

For example:

Processes (in order of arrival)	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF is *provably optimal*, in that it gives the minimum average waiting time for a given set of processes.

The Proof: essentially the same idea as the proof of the fact that OPT page-replacing algorithm is Optimal! Also, the same problem: in an online setting, when we don't know what processes are coming up, how do we compute future Burst Times?

Although SJF is optimal, it cannot be implemented at the level of CPU scheduling; there is no way to know the length of the next CPU burst.

What did we do in the context of page-replacing? We approximated the future, by looking at the past: LRU.

We predict the value of the next Burst Time. We make the assumption that the next CPU burst will be similar in length to the previous ones.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.

Let t_n be the length of the n-th CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for $0 \leq \alpha \leq 1$, define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

This formula defines the **exponential average**. The value t_n contains the most recent information, and τ_n stores the past history (let $\tau_1 = t_1$).

The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and the recent history has no effect (current conditions are assumed to be transient).

If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).

The SJF algorithm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing.

The new process may have a shorter next CPU burst than what is left of the currently executing process.

Preemptive SJF scheduling is called **shortest-remaining-time-first** scheduling.

The SJF algorithm is a special case of the general **priority-scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

The SJF algorithm is simply a priority algorithm where the assigned priority is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority.

Priorities

In general, process priorities can be defined either internally or externally.

Internal: time limits, memory requirements, nr of open files, etc.

External: importance of a process, often political in nature.

Priority scheduling can be preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

A preemptive algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

A nonpreemptive algorithm will put the new process at the head of the ready queue.

Problem with priority scheduling: **Starvation**. A process has a very low priority, and is never scheduled.

Solution: **Aging**. Gradually increase the priority of the processes that wait in the system for a long time.