Operating Systems
Lecture No. 14

Reading Material

- Chapter 6 of the textbook
- Lecture 14 on Virtual TV

Summary

- Basic concepts
- Scheduling criteria
- Preemptive and non-preemptive algorithms
- First-Come-First-Serve scheduling algorithm

Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes much wait until the CPU is free and can be rescheduled.
In multiprogramming, a process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. Multiprogramming entails productive usage of this time. When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process. Almost all computer resources are scheduled before use.

Life of a Process

As shown in Figure 14.1, process execution consists of a cycle of CPU execution and I/O wait. Processes alternates between these two states. Process execution begins with a CPU burst. An I/O burst follows that, and so on**.** Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.

An I/O bound program would typically have many very short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm. Figure 14.2 shows results on an empirical study regarding the CPU bursts of processes. The study shows that most of the processes have short CPU bursts of 2-3 milliseconds.

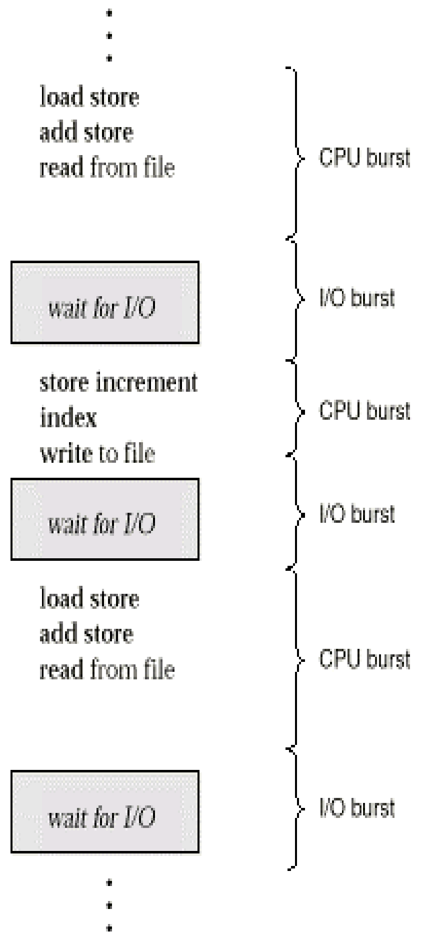


Figure 14.1 Alternating Sequence of CPU and I/O Bursts

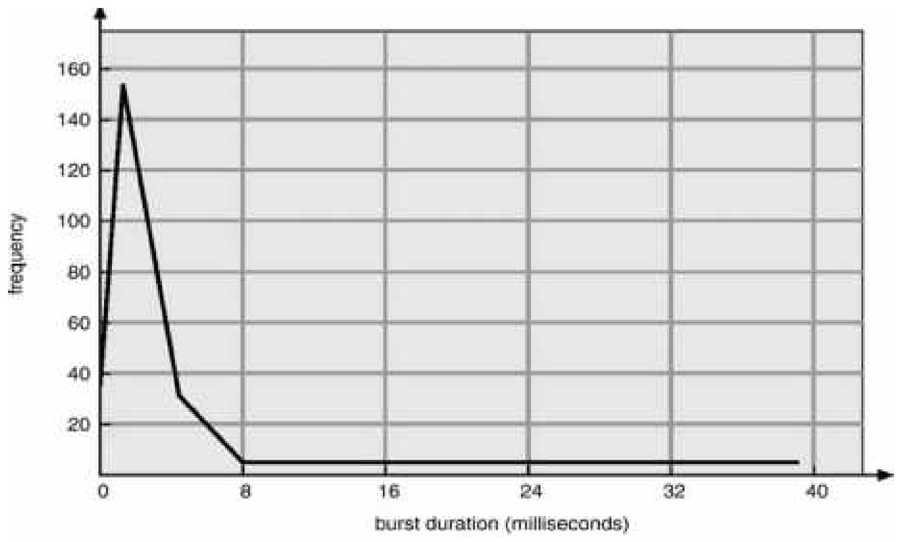


Figure 14.2 Histogram of CPU-burst Times

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in thc ready queue to be executcd. Thc **short-term scheduler** (i.c., thc CPU scheduler) selects a process to give it the CPU. It selects from among the processes in memory that are ready to execute, and invokes the dispatcher to have the CPU allocated to the selected process.

A ready queue can be implemented as a FIFO queue, a tree, or simply an unordered linked list. The records (nodes) in the ready queue are generally the process control blocks (PCBs) of processes.

Dispatcher

The **dispatcher** is a kernel module that takes control of the CPU from the current process and gives it to the process selected by the short-term scheduler. This function involves:

- Switching the context (i.e., saving the context of the current process and restoring the context of the newly selected process)
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Preemptive and Non-Preemptive Scheduling

CPU scheduling can take place under the following circumstances:

1. When a process switches from the running state to the waiting state (for example, an I/O request is being completed)
2. When a process switches from the running state to the ready state (for example when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In 1 and 4, there is no choice in terms of scheduling; a new process must be selected for execution. There is a choice in case of 2 and 3. When scheduling takes place only under 1 and 4, we say, scheduling is **non-preemptive**; otherwise the scheduling scheme is **preemptive**. Under non-preemptivc scheduling once the CPU has been allocated to a process the process keeps the CPU until either it switches to the waiting state, finishes its CPU burst, or terminates. This scheduling method does not require any special hardware needed for preemptive scheduling.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second proccss is run. Thc sccond proccss may try to rcad thc data, which arc currently in an inconsistent state. New mechanisms are needed to coordinate access to shared data. We discuss this topic in Chapter 7 of the textbook.

Scheduling Criteria

The scheduling criteria include:

- **CPU utilization**: We want to keep CPU as busy as possible. In a real system it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)
- **Throughput:** If CPU is busy executing processes then work is being done. One measure of work is the number of processes completed per time, called, **throughput.** We want to maximize the throughput.
- **Turnaround time:** The interval from the time of submission to the time of completion is the **turnaround time.** Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O. We want to minimize the turnaround time.
- **Waiting time:** Waiting time is the time spent waiting in the ready queue. We want to minimize the waiting time to increase CPU efficiency.
- **Response time:** It is the time from the submission of a request until the first response is produced. Thus response time is the amount of time it takes to start responding but not the time it takes to output that response. Response time should be minimized.

Scheduling Algorithms

We will now discuss some of the commonly used short-term scheduling algorithms. Some of these algorithms are suited well for batch systems and others for time-sharing systems. Here are the algorithms we will discuss:

- First-Come-First-Served (FCFS) Scheduling
- Shorted Job First (SJF) Scheduling
- Shortest Remaining Time First (SRTF) Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queues Scheduling
- Multilevel Feedback Queues Scheduling
- UNIX System V Scheduling

First-Come, First-Served (FCFS) Scheduling

The process that requests the CPU first (i.e., enters the ready queue first) is allocated the CPU first. The implementation of an FCFS policy is managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When CPU is free, it is allocated to the process at the head of the queue. The running process is removed from the queue. The average waiting time under FCFS policy is not minimal and may vary substantially if the process CPU-burst times vary greatly. FCFS is a non-preemptive scheduling algorithm.

We use the following system state to demonstrate the working of this algorithm. For simplicity, we assume that processes are in the ready queue at time 0.

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that processes arrive into the system in the order: P1, P2, P3. Processes are served in the order: P1, P2, P3.The **Gantt chart** for the schedule is shown in Figure 14.3.

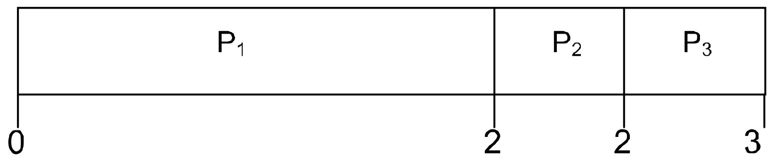


Figure 14.3 Gantt chart showing execution of processes in the order P1, P2, P3

Here are the waiting times for the three processes and the average waiting time per process.

- Waiting times P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0+24+27)/3 = 17

Suppose that processes arrive in the order: P2, P3, P1. The Gantt chart for the schedule is shown in Figure 14.4:

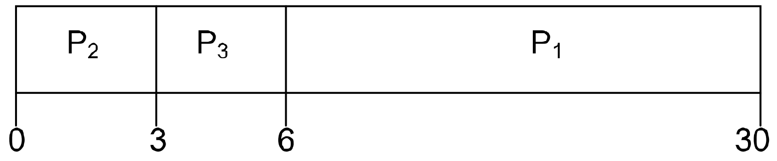


Figure 14.4 Gantt chart showing execution of processes in the order P2, P3, P1

Here are the waiting times for the three processes and the average waiting time per process.

- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

When FCFS scheduling algorithm is used, the **convoy effect** occurs when short processes wait behind a long process to use the CPU and enter the ready queue in a convoy after completing their I/O. This results in lower CPU and device utilization than might be possible if shorter processes were allowed to go first.

In the next lecture, we will discuss more scheduling algorithms.

Operating Systems
Lecture No. 15

Reading Material

- Chapter 6 of the textbook
- Lecture 15 on Virtual TV

Summary

- Scheduling algorithms

Shortest-Job-First Scheduling

This algorithm associates with ach process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. The real difficulty with the SJF algorithm is in knowing the length of the next CPU request. For long term scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.

For short-term CPU scheduling, there is no way to length of the next CPU burst. One approach is to try to approximate SJF scheduling, by assuming that the next CPU burst will be similar in length to the previous ones, for instance.

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 nth CPU burst and let τ_{n+1} be our predicted value for the next CPU burst. We define τ_{n+1} to be

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

where, $0 \leq \alpha \leq 1$. We discuss this equation in detail in a subsequent lecture.

The SJF algorithm may either be preemptive or non-preemptive. 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. A preemptive SJF algorithm preempts the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

We illustrate the working of the SJF algorithm by using the following system state.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

The Gantt chart for the execution of the four processes using SJF is shown in Figure 15.1.

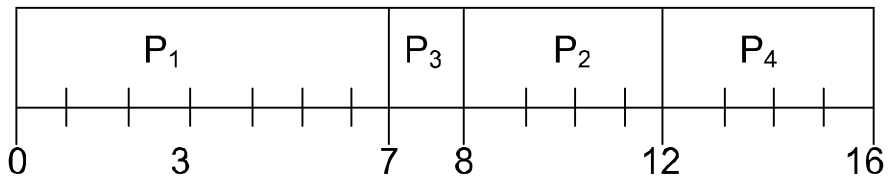


Figure 15.1 Gantt chart showing execution of processes using SJF

Here is the average waiting time per process.

- Average waiting time = (0 + 6 + 3 + 7)/4 = 4 time units

We illustrate the working of the SRTF algorithm by using the system state shown above. The Gantt chart for the execution of the four processes using SRTF is shown in Figure 15.2.

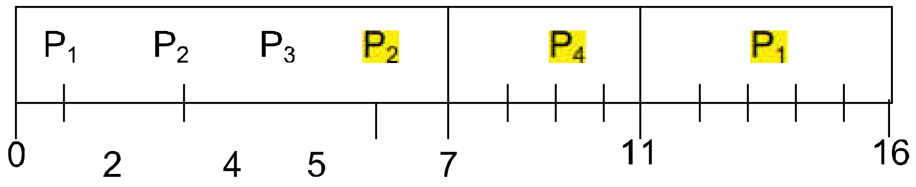


Figure 15.2 Gantt chart showing execution of processes using SRTF

Here is the average waiting time per process.

- Average waiting time = (9 + 1 + 0 +2)/4 = 3 time units

Priority Scheduling

SJF 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 (smallest integer ≡ highest priority). Equal priority processes are scheduled in FCFS order. The SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst of a process, the lower its priority, and vice versa.

Priority scheduling can either be preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority- scheduling algorithm will simply put the new process at the head of ready queue.

A major problem with priority- scheduling algorithms is **indefinite blocking** (or **starvation**). A process that is ready to run but lacking the CPU can be considered blocked-waiting for the CPU. A priority-scheduling algorithm can leave some low priority processes waiting indefinitely for the CPU. Legend has it that when they were phasing out IBM 7094 at MIT in 1973, they found a process stuck in the ready queue since 1967!

Aging is solution to the problem of indefinite blockage of low-priority processes. It involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priority numbers range from 0 (high priority) to 127 (high priority), we could decrement priority of every process periodically (say every 10 minutes). This would result in every process in the system eventually getting the highest priority in a reasonably short amount of time and scheduled to use the CPU.

Operating Systems
Lecture No. 16

Reading Material

- Chapter 6 of the textbook
- Lecture 16 on Virtual TV

Summary

- Scheduling algorithms

Why is SJF optimal?

SJF is an optimal algorithm because it decreases the wait times for short processes much more than it increases the wait times for long processes. Let's consider the example shown in Figure 16.1, in which the next CPU bursts of P1, P2, and P3 are 5, 3, and 2, respectively. The first Gantt chart shows execution of processes according to the longest-job-first algorithm, resulting in the waiting times for P1, P2, and P3 to be 0, 5, and 8 times units. The second Gantt chart shows execution of processes according to the shortest-job-first algorithm, resulting in the waiting times for P1, P2, and P3 to be 0, 2, and 5. Note that the waiting time for P2 has decreased from 5 to 2 and that of P3 has decreased from 8 to 0. The increase in the wait time for P1 is from 0 to 5, which is much smaller than the decrease in the wait times for P2 and P3.

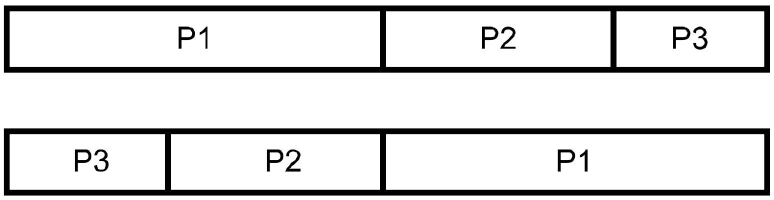


Figure 16.1 Two execution sequences for P1, P2, and P3: longest-job-first and shortest-job-first

Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling but preemption is added to switch between processes. A small unit of time, called a **time quantum (or time slice)** is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, we keep ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and then dispatches the process. One of the two things will then happen. The process may have a CPU burst of less than 1 time quantum, in which case the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of currently running process is longer than one time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will happen, the current process will be put at the tail of the ready queue, and the newly scheduled process will be given the CPU.

The average waiting time under the RR policy however is often quite long. It is a preemptive scheduling algorithm. If there are *n* processes n the ready queue, context switch time is t_{cs} and the time quantum is *q* then each process gets 1/*n* of the CPU time in chunks of at most *q* time units. Each process must wait no longer than $(n-1)*(q+t_{cs})$ time units until its next time quantum.

The performance of RR algorithm depends heavily on the size of the time quantum. If the time quantum is very large (infinite), the RR policy remains the same as the FCFS policy. If the time quantum is very small, the RR approach is called the **processor sharing** and appears to the users as though each of *n* processes has its own processor running at 1/*n* the speed of real processor (q must be large with respect to context switch, otherwise the overhead is too high). The drawback of small quantum is more frequent context switches. Since context switching is the cost of the algorithm and no useful work is done for any user process during context switching, the number of context switches should be minimized and the quantum should be chosen such that the ratio of a quantum to context switching is not less than 10:1 (i.e., context switching overhead should not be more than 10% of the time spent on doing useful work for a user process). Figure 16.2 shows increase in the number of context switches with decrease in quantum size.

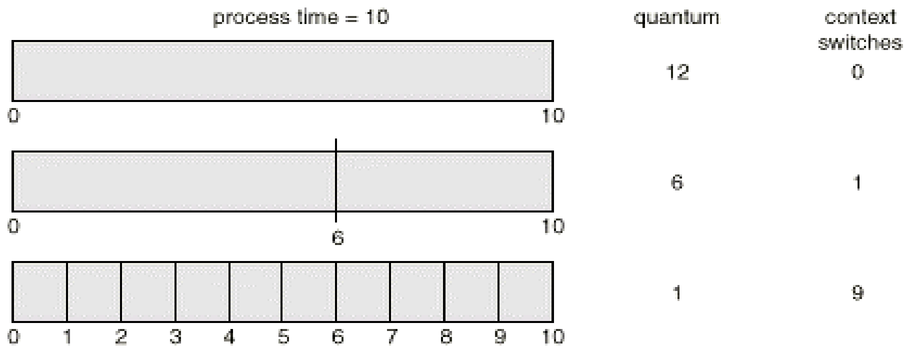
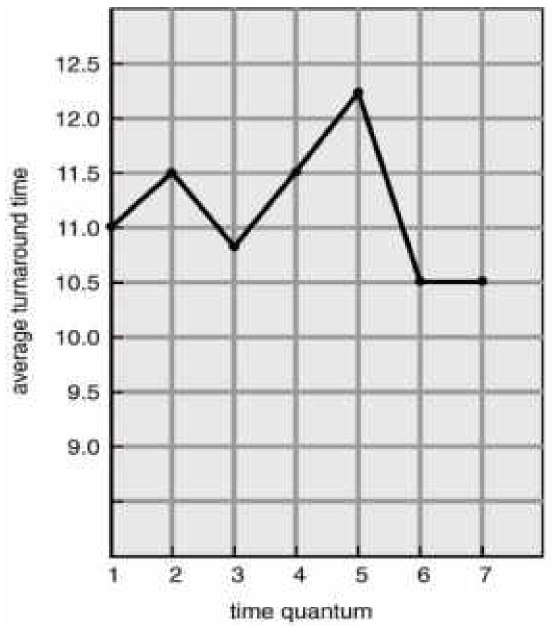


Figure 16.2 Quantum size versus number of context switches

The turnaround time of a process under round robin is also depends on the size of the time quantum. In Figure 16.3 we show a workload of four processes P1, P2, P3, and P4 with their next CPU bursts as 6, 3, 1, and 7 time units. The graph in the figure shows that best (smallest) turnaround time is achieved when quantum size is 6 or greater. Note that most of the given processes finish their next CPU bursts with quantum of 6 or greater. We can make a general statement that the round-robin algorithm gives smallest average turnaround time when quantum value is chosen such that most of the processes finish their next CPU bursts within the quantum.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 16.3 Turnaround time versus quantum size

We now consider the following system workload to illustrate working of the round-robin algorithm. Execution of P1 though P4 with quantum 20 is shown in Figure 16.4. In the table, original CPU bursts are shown in bold and remaining CPU bursts (after a process has used the CPU for one quantum) are shown in non-bold font.

Process	Burst Time
P1	**53** — 33 — 13
P2	**17**
P3	**68** — 48 — 28 — 8
P4	**24** — 4

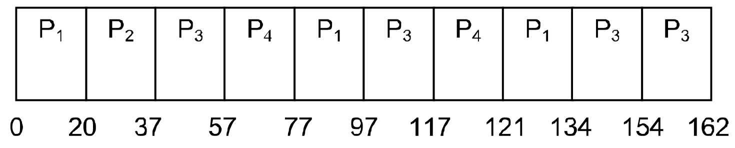


Figure 16.4 Gantt chart showing execution of P1, P2, P3, and P4 with quantum 20 time units

Figure 16.5 shows wait and turnaround times for the four processes. The average wait time for a process comes out to be 73 time units for round robin and 38 for SJF. Typically, RR has a higher average turnaround than SJF, but better response. In time-sharing systems, shorter response time for a process is more important than shorter turnaround time for the process. Thus, round-robin scheduler matches the requirements of time-sharing systems better than the SJF algorithm. SJF scheduler is better suited for batch systems, in which minimizing the turnaround time is the main criterion.

Process	Turnaround Time	Waiting Time
P1	134	134 – 53 = **81**
P2	37	37 – 17 = **20**
P3	162	162 – 68 = **94**
P4	121	121 – 24 = **97**

Figure 16.5 Wait and turnaround times for processes

Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (or **interactive**) processes and **background** (or **batch**) **processes**. These two types of processes have different response time requirements and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A **multilevel queue-scheduling algorithm** partitions the ready queue into several separate queues, as shown in Figure 16.5. Each queue has its own priority and scheduling algorithm. Processes are permanently assigned to one queue, generally based o some property of the process, such as memory size, process priority or process type. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling i.e., serve all from foreground then from background. Another possibility is to time slice between queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue, e.g., 80% to foreground in RR and 20% to background in FCFS. Scheduling across queues prevents starvation of processes in lower-priority queues.

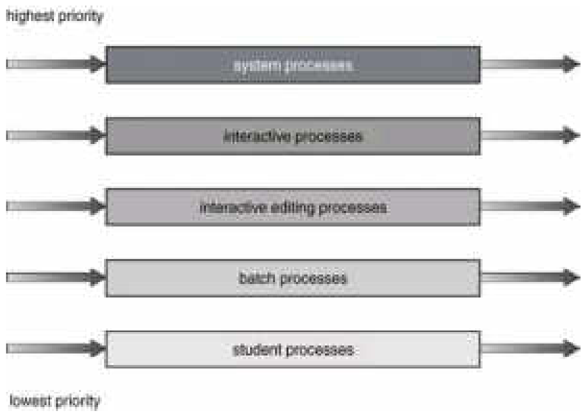


Figure 16.5 Multilevel queues scheduling

Operating Systems
Lecture No. 17

Reading Material
- Chapter 6 of the textbook
- Lecture 16 on Virtual TV

Summary
- Scheduling algorithms
- UNIX System V scheduling algorithm
- Optimal scheduling
- Algorithm evaluation

Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling allows a process to move between queues. The idea is to separate processes with different CPU burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues. Similarly a process that waits too long in a lower-priority queue may be moved o a higher priority queue. This form of aging prevents starvation.

In general, a multi-level feedback queue scheduler is defined by the following parameters:

- Number of queues
- Scheduling algorithm for each queue
- Method used to determine when to upgrade a process to higher priority queue
- Method used to determine when to demote a process
- Method used to determine which queue a process enters when it needs service

Figure 17.1 shows an example multilevel feedback queue scheduling system with the ready queue partitioned into three queues. In this system, processes with next CPU bursts less than or equal to 8 time units are processed with the shortest possible wait times, followed by processes with CPU bursts greater than 8 but no greater than 16 time units. Processes with CPU greater than 16 time units wait for the longest time.

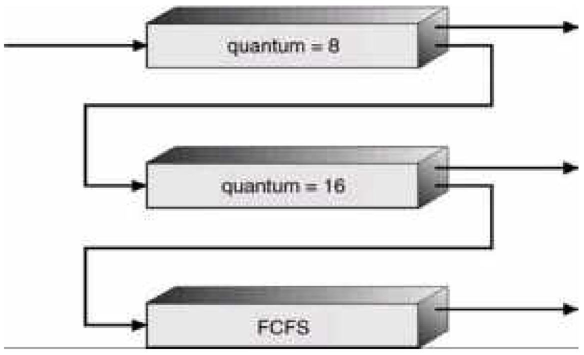


Figure 17.1 Multilevel Feedback Queues Scheduling

UNIX System V scheduling algorithm

UNIX System V scheduling algorithm is essentially a multilevel feedback priority queues algorithm with round robin within each queue, the quantum being equal to1 second. The priorities are divided into two groups/bands:

- Kernel Group
- User Group

Priorities in the Kernel Group are assigned in a manner to minimize bottlenecks, i.e, processes waiting in a lower-level routine get higher priorities than those waiting at relatively higher-level routines. We discuss this issue in detail in the lecture with an example. In decreasing order of priority, the kernel bands are:

- Swapper
- Block I/O device control processes
- File manipulation
- Character I/O device control processes
- User processes

The priorities of processes in the Kernel Group remain fixed whereas the priorities of processes in the User Group are recalculated every second. Inside the User Group, the CPU-bound processes are penalized at the expense of I/O-bound processes. Figure 17.2 shows the priority bands for the various kernel and user processes.

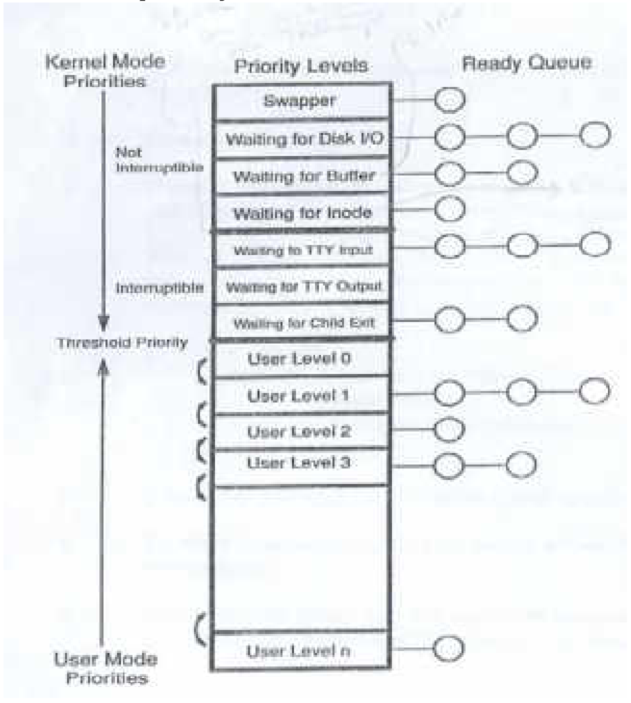


Figure 17.2. UNIX System V Scheduling Algorithm

Every second, the priority number of all those processes that are in the main memory and ready to run is updated by using the following formula:

Priority # = (Recent CPU Usage)/2 + Threshold Priority + nice

The ‘threshold priority’ and ‘nice’ values are always positive to prevent a user from migrating out of its assigned group and into a kernel group. You can change the nice value of your process with the `nice` command.

In Figure 17.3, we illustrate the working of the algorithm with an example. Note that recent CPU usage of the current process is updated every clock tick; we assume that clock interrupt occurs every sixtieth of a second. The priority number of every process in the ready queue is updated every second and the decay function is applied before recalculating the priority numbers of processes.

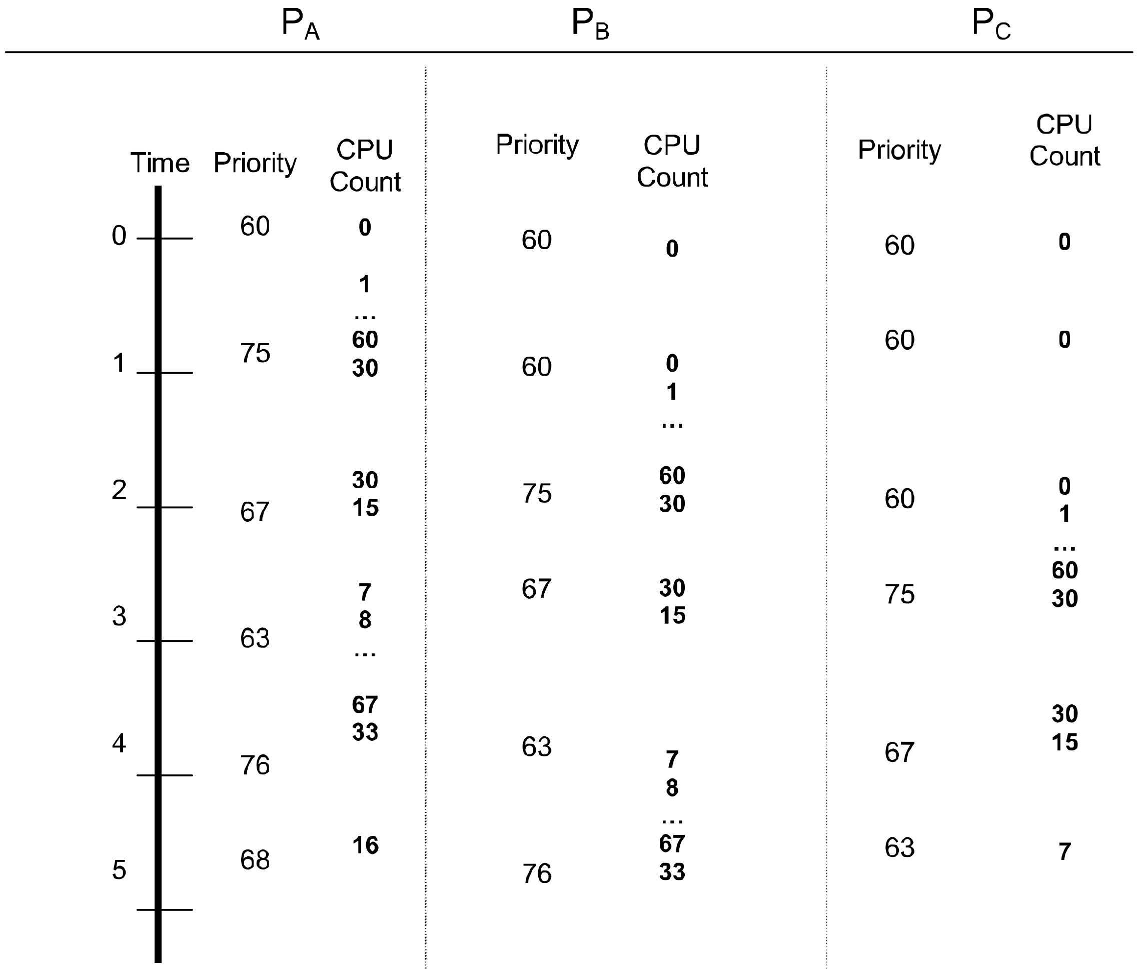


Figure 17.3 Illustration of the UNIX System V Scheduling Algorithm

Figure 17.4 shows that in case of a tie, processes are scheduled on First-Come-First-Serve basis.

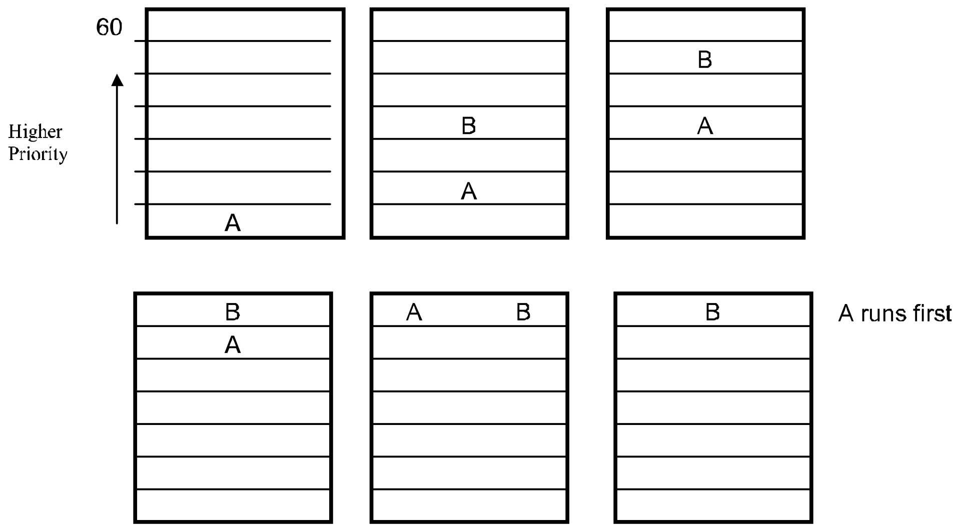


Figure 17.4 FCFS Algorithm is Used in Case of a Tie

Algorithm Evaluation

To select an algorithm, we must take into account certain factors, defining their relative importance, such as:

- Maximum CPU utilization under the constraint that maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Scheduling algorithms can be evaluated by using the following techniques:

Analytic Evaluation

A scheduling algorithm and some system workload are used to produce a formula or number, which gives the performance of the algorithm for that workload. Analytic evaluation falls under two categories:

Deterministic modeling

Deterministic modeling is a type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for workload in terms of numbers for parameters such as average wait time, average turnaround time, and average response time. Gantt charts are used to show executions of processes. We have been using this technique to explain the working of an algorithm as well as to evaluate the performance of an algorithm with a given workload.

Deterministic modeling is simple and fast. It gives exact numbers, allowing the algorithms to be compared. However it requires exact numbers for input and its answers apply to only those cases.

Queuing Models

The computer system can be defined as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as are I/O systems with their device queues. Knowing the arrival and service rates of processes for various servers, we can compute utilization, average queue length, average wait time, and so on. This kind of study is called **queuing-network analysis**. If n is the average queue length, W is the average waiting time in the queue, and let λ is the average arrival rate for new processes in the queue, then

$$n = \lambda * W$$

This formula is called the **Little's formula**, which is the basis of **queuing theory**, a branch of mathematics used to analyze systems involving queues and servers.

At the moment, the classes of algorithms and distributions that can be handled by queuing analysis are fairly limited. The mathematics involved is complicated and distributions can be difficult to work with. It is also generally necessary to make a number of independent assumptions that may not be accurate. Thus so that they will be able to compute an answer, queuing models are often an approximation of real systems. As a result, the accuracy of the computed results may be questionable.

The table in Figure 17.5 shows the average waiting times and average queue lengths for the various scheduling algorithms for a pre-determined system workload, computed by using Little's formula. The average job arrival rate is 0.5 jobs per unit time.

Algorithm	Average Wait Time $W = t_w$	Average Queue Length (n)
FCFS	4.6	2.3
SJF	3.6	1.8
SRTF	3.2	1.6
RR (q=1	7.0	3.5
RR (q=4)	6.0	3.0

Figure 17.5 Average Wait Time and Average Queue Length Computed With Little's Equation

Simulations

Simulations involve programming a model of the computer system, in order to get a more accurate evaluation of the scheduling algorithms. Software date structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed. Figure 17.6 shows the schematic for a simulation system used to evaluate the various scheduling algorithms.

Some of the major issues with simulation are:

- Expensive: hours of programming and execution time are required
- Simulations may be erroneous because of the assumptions about distributions used for arrival and service rates may not reflect a real environment

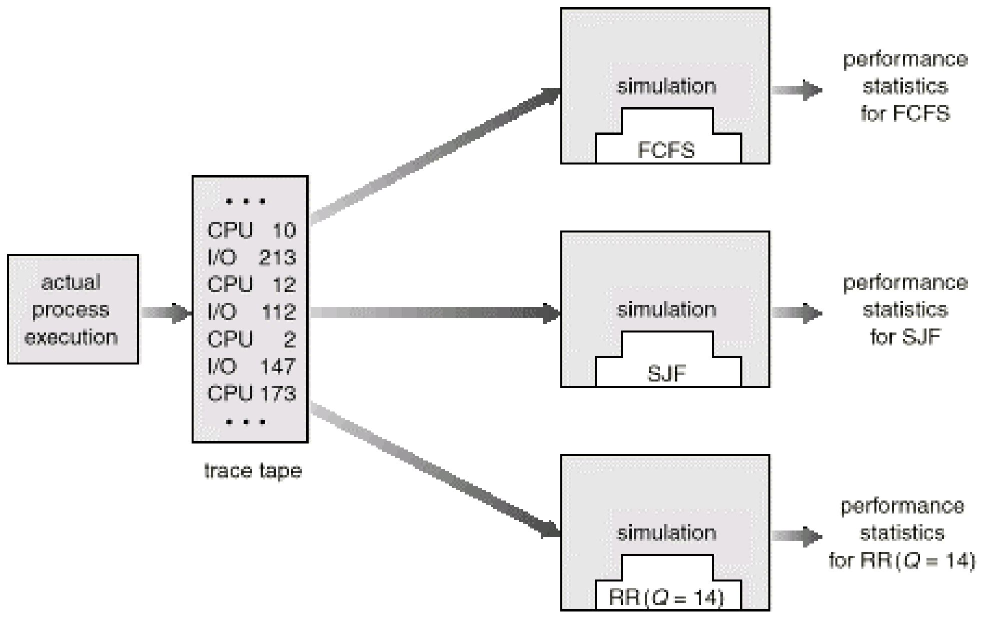


Figure 17.6 Simulation of Scheduling Algorithms

Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The Open Source software licensing has made it possible for us to test various algorithms by implementing them in the Linux kernel and measuring their true performance.

The major difficulty is the cost of this approach. The expense is incurred in coding the algorithm and modifying the operating system to support it, as well as its required data structures. The other difficulty with any algorithm evaluation is that the environment in which the algorithm works will change.

Operating Systems
Lecture No. 18 and 19

Reading Material

- Chapter 7 of the textbook
- Lectures 18 and 19 on Virtual TV

Summary

- Process Synchronization: the basic concept
- The Critical Section Problem
- Solutions for the Critical Section Problem
- 2-Process Critical Section Problem solutions

Process Synchronization

Concurrent processes or threads often need access to shared data and shared resources. If there is no controlled access to shared data, it is often possible to obtain an inconsistent state of this data. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes, and hence various process synchronization methods are used. In the producer-consumer problem that was discussed earlier, the version only allows one item less than the buffer size to be stored, to provide a solution for the buffer to use its entire capacity of N items is not simple. The producer and consumer share data structure 'buffer' and use other variables shown below:

```
#define BUFFER_SIZE 10
typedef struct
{
   ...
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

The code for the producer process is:

```
while(1)
{
   /*Produce an item in nextProduced*/
   while(counter == BUFFER_SIZE); /*do nothing*/
   buffer[in]=nextProduced;
   in=(in+1)%BUFFER_SIZE;
   counter++;
}
```

The code for the consumer process is:

```
while(1)
{
   while(counter==0); //do nothing
   nextConsumed=buffer[out];
   out=(out+1)%BUFFER_SIZE;
   counter--;
   /*Consume the item in nextConsumed*/
}
```

Both producer and consumer routines may not execute properly if executed concurrently. Suppose that the value of the counter is 5, and that both the producer and the consumer execute the statement counter++ and counter- - concurrently. Following the execution of these statements the value of the counter may be 4,5,or 6! The only correct result of these statements should be counter= =5, which is generated if the consumer and the producer execute separately. Suppose counter++ is implemented in machine code as the following instructions:

```
MOV  R1, counter
INC  R1
MOV  counter, R1
```

whereas counter- - maybe implemented as:

```
MOV  R2, counter
DEC  R2
MOV  counter, R2
```

If both the producer and consumer attempt to update the buffer concurrently, the machine language statements may get interleaved. Interleaving depends upon how the producer and consumer processes are scheduled. Assume counter is initially 5. One interleaving of statements is:

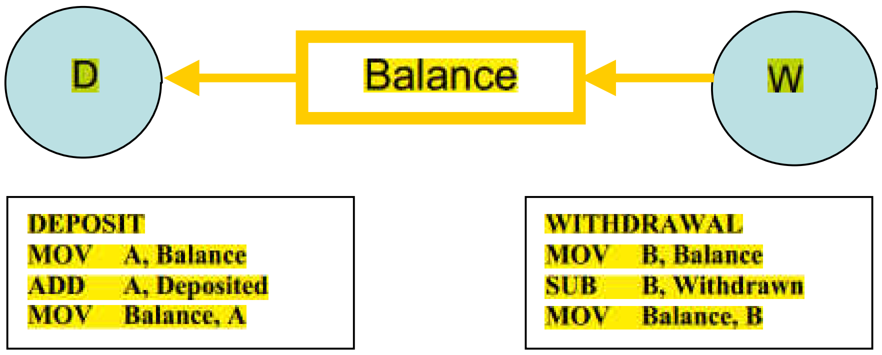
```
producer: MOV  R1, counter    (R1 = 5)
          INC  R1             (R1 = 6)
consumer: MOV  R2, counter    (R2 = 5)
          DEC  R2             (R2 = 4)
producer: MOV  counter, R1    (counter = 6)
consumer: MOV  counter, R2    (counter = 4)
```

The value of `count` will be 4, where the correct result should be 5. The value of `count` could also be 6 if producer executes `MOV counter, R1` at the end. The reason for this state is that we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the manipulation depends on the particular order in which the access takes place, is called a **race condition**. To guard against such race conditions, we require synchronization of processes.

Concurrent transactions in a bank or in an airline reservation (or travel agent) office are a couple of other examples that illustrates the critical section problem. We show interleaving of two bank transactions, a deposit and a withdrawal. Here are the details of the transactions:

- Current balance = Rs. 50,000
- Check deposited = Rs. 10,000
- ATM withdrawn = Rs. 5,000

The codes for deposit and withdrawal are shown in Figure 18.1.



```
DEPOSIT
MOV    A, Balance
ADD    A, Deposited
MOV    Balance, A
```

```
WITHDRAWAL
MOV    B, Balance
SUB    B, Withdrawn
MOV    Balance, B
```

Figure 18.1 Bank transactions—deposit and withdrawal

Here is what may happen if the two transactions are allowed to execute concurrently, i.e., the transactions are allowed to interleave. Note that in this case the final balance will be Rs. 45,000, i.e., a loss of Rs. 5,000. If `MOV Balance,A` executes at the end, the result will be a gain of Rs. 5,000. In both cases, the final result is wrong.

```
Check Deposit:
        MOV   A, Balance          // A = 50,000
        ADD   A, Deposited        // A = 60,000
ATM Withdrawal:
        MOV   B, Balance          // B = 50,000
        SUB   B, Withdrawn        // B = 45,000
Check Deposit:
        MOV   Balance, A          // Balance = 60,000
ATM Withdrawal:
        MOV   Balance, B          // Balance = 45,000
```

The Critical Section Problem

Critical Section: A piece of code in a cooperating process in which the process may updates shared data (variable, file, database, etc.).

Critical Section Problem: Serialize executions of critical sections in cooperating processes.

When a process executes code that manipulates shared data (or resource), we say that the process is in its critical section (for that shared data). The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors). So each process must first request permission to enter its critical section. The section of code implementing this request is called the **entry section.** The remaining code is the **remainder section.** The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

There can be three kinds of solution to the critical section problem:

- Software based solutions
- Hardware based solutions
- Operating system based solution

We discuss the software solutions first. Regardless of the type of solution, the structure of the solution should be as follows. The Entry and Exist sections comprise solution for the problem.

```
do
{
        Entry section

          critical section

        Exit section

          remainder section

} while(1)
```

Solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three requirements:

1. **Mutual Exclusion**
   If process P_i is executing in its critical section, then no other process can be executing in their critical section.

2. **Progress**
   If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded Waiting**
   There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Assumptions

While formulating a solution, we must keep the following assumptions in mind:

- Assume that each process executes at a nonzero speed
- No assumption can be made regarding the relative speeds of the N processes.

2-Process Solutions to the Critical Section Problem

In this section algorithms that are applicable to two processes will be discussed. The processes are P_0 and P_1. When presenting P_i, we use P_j to denote the other process. An assumption is that the basic machine language instructions such as load and store are executed atomically, that is an operation that completes in its entirety without interruption.

Algorithm 1

The first approach is to let the processes share a common integer variable **turn** initialized to 0 or 1. If turn = = i, then process P_i is allowed to execute in its critical section. The structure of the process P_i is as follows:

```
do
{
    while(turn!=j);

      critical section

     turn=j;

      remainder section
} while(1)
```

This solution ensures mutual exclusion, that is only one process at a time can be in its critical section. However it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if turn= =0 and P_1 is ready to enter its critical section, P_1 cannot do so even though P_0 may be in its remainder section. The bounded wait condition is satisfied though, because there is an alternation between the turns of the two processes.

Algorithm 2

In algorithm two, the variable turn is replaced with an array `boolean flag[2]` whose elements are initialized to false. If flag is true for a process that indicates that the process is ready to enter its critical section. The structure of process P_i is shown:

```
do
{
        flag[i]=true;
        while(flag[j]);

                critical section

        flag[i]=false;

                remainder section
} while(1)
```

In this algorithm P_i sets `flag[i]= true` signaling that it is ready to enter its critical section. Then P_i checks to verify that process P_j is not also ready to enter its critical section. If P_j were ready, then P_i would wait until P_j had indicated that it no longer needed to be in the critical section (that is until `flag[j]=false`). At this point P_i would enter the critical section. On exiting the critical section, P_i would set `flag[i]=false` allowing the other process to enter its critical section. In this solution, the mutual exclusion requirement is satisfied. Unfortunately the progress condition is not met; consider the following execution sequence:

T_0: P_0 sets `flag[0]= true`
T_1: P_1 sets `flag[1]= true`

Now both the processes are looping forever in their respective while statements.

Operating Systems
Lecture No. 20

Reading Material

- Chapter 7 of the textbook
- Lecture 20 on Virtual TV

Summary

- 2-Process Critical Section Problem (continued)
- n-Process Critical Section Problem
- The Bakery Algorithm

2-Process Critical Section Problem (continued)

We discussed two solutions for the 2-process critical section problem in lecture 19 but both were not acceptable because they did not satisfy the progress condition. Here is a good solution for the critical section problem that satisfies all three requirements of a good solution.

Algorithm 3

The processes share two variables:

```
boolean flag[2];
int turn;
```

The boolean array of 'flag' is initialized to false, whereas 'turn' maybe 0 or 1. The structure of the process is as follows:

```
do
{
          flag[i]=true;
          turn=j;
          while(flag[j] && turn==j);

             critical section

          flag[i]=false;

             remainder section

} while(1)
```

To enter its critical section, P_i sets flag[i] to true, and sets 'turn' to j, asserting that if the other process wishes to enter its critical section, it may do so. If both try to enter at the same time, they will attempt to set 'turn' to i and j. However, only one of these assignments will last, the other will occur but be overwritten instantly. Hence, the eventual value of 'turn' will decide which process gets to enter its critical section.

To prove mutual exclusion, note that P_i enters its critical section only if either flag[j]=false or turn=i. Also, if both processes were executing in their critical sections at the same time, then flag[0]= = flag[1]= = true. These two observations suggest that P_0 and P_1 could not have found both conditions in the while statement true at the same time, since the value of 'turn' can either be 0 or 1. Hence only one process say P_0 must have successfully exited the while statement. Hence mutual exclusion is preserved.

To prove bounded wait and progress requirements, we note that a process P_i can be prevented the critical section only if it is stuck in the while loop with the condition flag[j]= =true and turn=j. If P_j is not ready to enter the critical section, then flag[j]=flase and P_i can enter its critical section. If P_j has set flag[j]=true and is also executing its while statement then either turn=i or turn=j. If turn=i then P_i enters its critical section, otherwise P_j. However, whenever a process finishes executing in its critical section, lets assume P_j, it resets flag[j] to false allowing P_i to enter its critical section. If P_j resets flag[j]=true, then it must also set 'turn' to i, and since P_i does not change the value of 'turn' while executing in its while statement, P_i will enter its critical section (progress) after at most one entry by P_j (bounded waiting).

N-Process Critical Section Problem

In this section we extend the critical section problem of two processes to include n processes. Consider a system of n processes (P_o, P_1 ...... P_{n-1}). Each process has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing a file and so on. The important feature of the system in that, when one process enters its critical section, no other process is allowed to execute in its critical section. Thus the execution of critical sections by the processes is mutually exclusive in time. The critical section problem is to design a protocol to serialize executions of critical sections. Each process must request permission to enter its critical section. Many solutions are available in the literature to solve the N-process critical section problem. We will discuss a simple and elegant solution, known as the Bakery algorithm.

The Bakery Algorithm

The bakery algorithm is due to Leslie Lamport and is based on a scheduling algorithm commonly used in bakeries, ice-cream stores, and other locations where order must be made out of chaos. On entering the store, each customer receives a number. The customer with the lowest number is served next. Before entering its critical section, process receives a ticket number. Holder of the smallest ticket number enters its critical section. Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) will not receive the same number. In the case of a tie, the process with the lowest ID is served first. If processes Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first. The ticket numbering scheme always generates numbers in the increasing order of enumeration; i.e., 1, 2, 3, 4, 5 ...

Since process names are unique and totally ordered, our algorithm is completely deterministic. The common data structures are:

```
boolean choosing [n];
int number[n];
```

Initially these data structures are initialized to false and 0, respectively. The following notation is defined for convenience:

- (ticket #, process id #)
- (a,b) < (c,d) if a<c or if a= =c and b<d.
- max($a_0, \ldots a_{n-1}$) is a number, k, such that k>= a_i for i=0,…n-1

The structure of process P_i used in the bakery algorithm is as follows:

```
do
{
  choosing[i] = true;
  number[i] = max(number[0],number[1],..number[n-1])+1;
  choosing[i] = false;

  for(j=0; j<n; j++) {
    while(choosing[j]);
    while((number[j]!=0) && ((number[j],j) < (number[i],i)));
  }
             Critical section
  number[i]=0;
             Remainder section
} while(1);
```

To prove that the bakery algorithm is correct, we need to first show that if P_i is in its critical section and P_k has already chosen its number k!=0, then ((number [i],i) < (number[k],k)). Consider P_i in its critical section and P_k trying to enter its critical section. When process P_k executes the second while statement for j= = i it finds that,

- number[i] != 0
- (number[i],i) < (number[k],k)

Thus it keeps looping in the while statement until P_i leaves the P_i critical section. Hence mutual exclusion is preserved. For progress and bounded wait we observe that the processes enter their critical section on a first come first serve basis.

Following is an example of how the Bakery algorithm works. In the first table, we show that there are five processes, P0 through P4. P1's number is 0 because it is not interested in getting into its critical section at this time. All other processes are interested in entering their critical sections and have chosen non-zero numbers by using the `max()` function in their entry sections.

Process	Number
P0	3
P1	0
P2	7
P3	4
P4	8

The following table shows the status of all the processes as they execute the 'for' loops in their entry sections. The gray cells show processes waiting in the second while loops in their entry sections. The table shows that P0 never waits for any process and is, therefore, the first process to enter its critical section, while all other processes wait in their second while loops for j = = 0, indicating that they are waiting for P0 to get out of its critical section and then they would make progress (i.e., they will get out the while loop, increment j by one, and continue their execution).

You can make the following observations by following the Bakery algorithm closely with the help of this table:

- P1 not interested to get into its critical section ⇒ number[1] is 0
- P2, P3, and P4 wait for P0
- P0 gets into its CS, get out, and sets its number to 0
- P3 get into its CS and P2 and P4 wait for it to get out of its CS
- P2 gets into its CS and P4 waits for it to get out
- P4 gets into its CS
- Sequence of execution of processes: <P0, P3, P2, P4>

j	**P0**	**P2**	**P3**	**P4**
0	(3,0) < (3,0)	(3,0) < (7,2)	(3,0) < (4,3)	(3,0) < (8,4)
1	Number[1] = 0	Number[1] = 0	Number[1] = 0	Number[1] = 0
2	(7,2) < (3,0)	(7,2) < (7,2)	(7,2) < (4,3)	(7,2) < (8,4)
3	(4,3) < (3,0)	(4,3) < (7,2)	(4,3) < (4,3)	(4,3) < (8,4)
4	(8,4) < (3,0)	(8,4) < (7,2)	(8,4) < (4,3)	(8,4) < (8,4)

Operating Systems
Lecture No. 21

Reading Material

- Chapter 7 of the textbook
- Lecture 21 on Virtual TV

Summary

- Hardware solutions

Hardware Solutions for the Critical Section Problem

In this section, we discuss some simple hardware (CPU) instructions that can be used to provide synchronization between processes and are available on many systems.

The critical section problem can be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately this solution is not feasible in a multiprocessing environment, as disabling interrupts can be time consuming as the message is passed to all processors. This message passing delays entry into each critical section, and system efficiency decreases.

Normally, access to a memory location excludes other accesses to that same location. Designers have proposed machine instructions that perform two operations atomically (indivisibly) on the same memory location (e.g., reading and writing). The execution of such an instruction is also mutually exclusive (even on Multiprocessors). They can be used to provide mutual exclusion but other mechanisms are needed to satisfy the other two requirements of a good solution to the critical section problem.

We can use these special instructions to solve the critical section problem. These instructions are TestAndSet (also known as TestAndSetLock; TSL) and Swap. The semantics of the TestAndSet instruction are as follows:

```
boolean TestAndSet(Boolean &target)
{
  boolean rv=target;
  target=true;
  return rv;
}
```

The semantics simply say that the instruction saves the current value of 'target', set it to true, and returns the saved value.

The important characteristic is that this instruction is executed atomically. Thus if two TestAndSet instructions are executed simultaneously, they will be executed sequentially in some arbitrary order.

If the machine supports TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i becomes:

```
do
{
        while (TestAndSet(lock)) ;
                Critical section
        lock=false;
                Remainder section
} while(1);
```

The above TSL-based solution is no good because even though mutual exclusion and progress are satisfied, bounded waiting is not.

The semantics of the Swap instruction, another atomic instruction, are, as expected, as follows:

```
boolean Swap(boolean &a, boolean &b)
{
  boolean temp=a;
  a=b;
  b=temp;
}
```

If the machine supports the Swap instruction, mutual exclusion can be implemented as follows. A global Boolean variable lock is declared and is initialized to false. In addition each process also has a local Boolean variable key. The structure of process P_i is:

```
do
{
        key=true;
        while(key == true)
            Swap(lock,key);
                Critical section
        lock=false;
                Remainder section
} while(1);
```

Just like the TSL-based solution shown in this section, the above Swap-based solution is not good because even though mutual exclusion and progress are satisfied, bounded waiting is not. In the next lecture, we will discuss a good solution for the critical section problem by using the hardware instructions.

Operating Systems
Lecture No. 22

Reading Material

- Chapter 7 of the textbook
- Lecture 22 on Virtual TV

Summary

- Hardware based solutions
- Semaphores
- Semaphore based solutions for the critical section problem

Hardware Solutions

In lecture 21 we started discussing the hardware solutions for the critical section problem. We discussed two possible solutions but realized that whereas both solutions satisfied the mutual exclusion and bounded waiting conditions, neither satisfied the progress condition. We now describe a solution that satisfies all three requirements of a solution to the critical section problem.

Algorithm 3

In this algorithm, we combine the ideas of the first two algorithms. The common data structures used by a cooperating process are:

```
boolean waiting[n];
boolean lock;
```

The structure of process P_i is:

```
do
{
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
            key = TestAndSet(lock);
        waiting[i] = false;

            Critical section

        j = (i+1) % n;
        while ((j!=i) && !waiting[j])
            j = (j+1)% n;
        if (j == i)
            lock = false;
        else
            waiting[j] = false;

            Remainder section

} while(1);
```

These data structures are initialized to false. To prove that the mutual exclusion requirement is met, we note that process P_i can enter its critical section only if either waiting[i]= = false or key = = false. The value of key can become false only if TestAndSet is executed. The first process to execute the TestAndSet instruction will find key= =false; all others must wait. The variable waiting[i] can only become false if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual exclusion requirement.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, …, n-1, 0, 1, …, i-1). It designates the first process it sees that is in its entry section with waiting[j]=true as the next one to enter its critical section. Any process waiting to do so will enter its critical section within n-1 turns.

Semaphores

Hardware solutions to synchronization problems are not easy to generalize to more complex problems. To overcome this difficulty we can use a synchronization tool called a semaphore. A **semaphore** S is an integer variable that, apart from initialization is accessible only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait) and V (for signal). The classical definitions of wait and signal are:

```
wait(S) {
  while(S<=0)
  ;// no op
  S--;
}
```

```
signal(S) {
  S++;
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process is updating the value of a semaphore, other processes cannot simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S (S<=0) and its possible modification (S--) must also be executed without interruption.

We can use semaphores to deal with the n-process critical section problem. The n processes share a semaphore, **mutex** (standing for mutual exclusion) initialized to 1. Each process P_i is organized as follows:

```
do
{
        wait(mutex);
              Critical section
        signal(mutex);
              Remainder section
} while(1);
```

As was the case with the hardware-based solutions, this is not a good solution because even though it satisfies mutual exclusion and progress, it does not satisfy bounded wait.

In a uni-processor environment, to ensure atomic execution, while executing wait and signal, interrupts can be disabled. In case of a multi-processor environment, to ensure atomic execution is one can lock the data bus, or use a soft solution such as the Bakery algorithm.

The main disadvantage of the semaphore discussed in the previous section is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. This is, spinlocks are useful when they are expected to be held for short times. The definition of semaphore should be modified to eliminate busy waiting. We will discuss the modified definition of semaphore in the next lecture.

Operating Systems
Lecture No. 23

Reading Material
- Chapter 7 of the textbook
- Lecture 23 on Virtual TV

Summary
- Busy waiting
- New definition of semaphore
- Process synchronization
- Problems with the use of semaphore: deadlock, starvation, and violation of mutual exclusion

Semaphores
The main disadvantage of the semaphore discussed in the previous section is that they all require **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. This, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of semaphore and the wait and signal operations on it. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU scheduling algorithm.)

Such an implementation of a semaphore is as follows:

```
typedef struct {
  int value;
  struct process *L;
} semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore; it is added to the list of processes. A signal operation removes one process from the list of the waiting processes and awakens that process. The wait operation can be defined as:

```
void wait(semaphore S) {
  S.value--;
  if(S.value < 0) {
     add this process to S.L;
     block();
 }
}
```

The signal semaphore operation can be defined as

```
void signal wait(semaphore S) {
  S.value++;
  if(S.value <= 0) {
     remove a process P from S.L;
     wakeup(P);
 }
}
```

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls. The negative value of S.value indicates the number of processes waiting for the semaphore. A pointer in the PCB needed to maintain a queue of processes waiting for a semaphore. As mentioned before, the busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Process Synchronization

You can use semaphores to synchronize cooperating processes. Consider, for example, that you want to execute statement B in Pj only after statement A has been executed in Pi. You can solve this problem by using a semaphore S initialized to 0 and structuring the codes for Pi and Pj as follows:

Pi	**Pj**
`...`	`...`
`A;`	`wait(S);`
`signal(S);`	`B;`
`...`	`...`

Pj will not be able to execute statement B until Pi has executed its statements A and signal(S).

Here is another synchronization problem that can be solved easily using semaphores. We want to ensure that statement S1 in P1 executes only after statement S2 in P2 has executed, and statement S2 in P2 should execute only after statement S3 in P3 has executed. One possible semaphore-based solution uses two semaphores, A and B. Here is the solution.

```
semaphore A=0, B=0;
P1             P2             P3

...            ...            ...
wait(A);       wait(B);       S3;
S1;            S2;            signal(B);
               signal(A);
...            ...            ...
```

Problems with Semaphores

Here are some key points about the use of semaphores:

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- The wait(S) and signal(S) operations are scattered among several processes. Hence, it is difficult to understand their effects.
- Usage of semaphores must be correct in all the processes.
- One bad (or malicious) process can fail the entire system of cooperating processes.

Incorrect use of semaphores can cause serious problems. We now discuss a few of these problems.

Deadlocks and Starvation

A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set. Here are a couple of examples of deadlocks in our daily lives.

- Traffic deadlocks
- One-way bridge-crossing

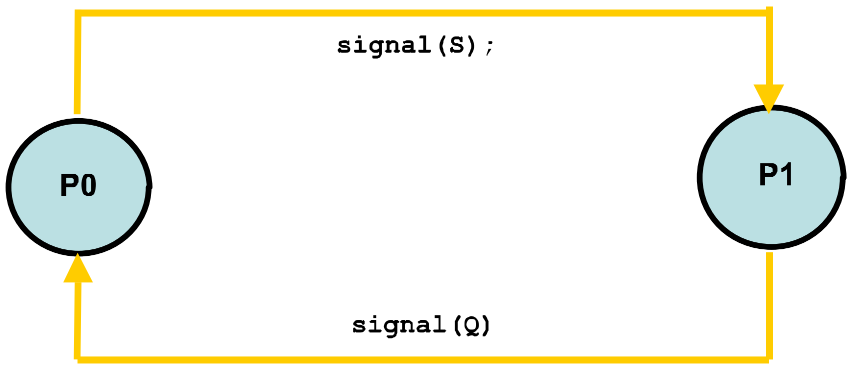
Starvation is infinite blocking caused due to unavailability of resources. Here is an example of a deadlock.

```
P0             P1

wait(S);       wait(Q);
wait(Q);       wait(S);
...            ...
signal(S);     signal(Q);
signal(Q);     signal(S);
...            ...
```

P0 and P1 need to get two semaphores, S and Q, before executing their critical sections. The following code structures can cause a deadlock involving P0 and P1. In this example, P0 grabs semaphore S and P1 obtains semaphore Q. Then, P0 waits for Q and P1 waits for S. P0 waits for P1 to execute signal(Q) and P1 waits for P0 to execute signal(S).

Neither process will execute the respective instruction—a typical deadlock situation. The following diagram shows the situation pictorially.



Here is an example of starvation. The code structures are self-explanatory.

```
P0              P1

wait(S);        wait(S);
...             ...
wait(S);        signal(S);
...             ...
```

Violation of Mutual Exclusion
In the following example, the principle of mutual exclusion is violated. Again, the code structures are self-explanatory. If you have any questions about them, please see the lecture video.

```
P0              P1

signal(S);      wait(S);
...             ...
wait(S);        signal(S);
...             ...
```

These problems are due to programming errors because of the tandem use of the wait and signal operations. The solution to these problems is higher-level language constructs such as critical region (region statement) and monitor. We discuss these constructs and their use to solve the critical section and synchronization problems in the next lecture.

Operating Systems
Lecture No. 24

Reading Material

- Chapter 7 of the textbook
- Lecture 24 on Virtual TV

Summary

- Counting semaphores
- Classical synchronization problems
- Bounded buffer problem
- Readers and writers problem
- Dining philosophers problem

Semaphores

There are two kinds of semaphores:

- **Counting semaphore** whose integer value can range over an unrestricted integer domain.
- **Binary semaphore** whose integer value cannot be > 1; can be simpler to implement.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
int C;
```

Initially S1=1, S2=0, and the value of integer C is set to the initial value of the counting semaphore S. The wait operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C--;
if(C < 0) {
   signal(S1);
   wait(S2);
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C++;
if(C <= 0)
  signal(S2);
else
  signal(S1);
```

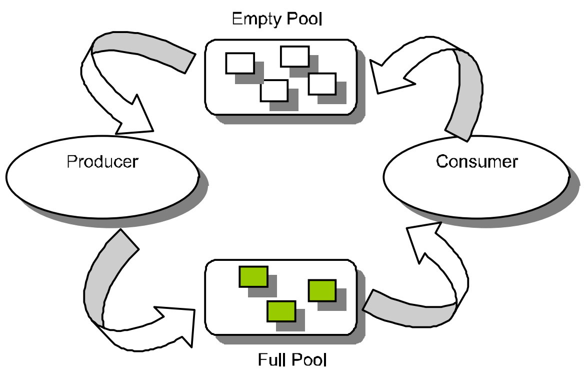
Classic Problems of Synchronization

The three classic problems of synchronization are:

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

The bounded-buffer problem, which was introduced in a previous lecture, is commonly used to illustrate the power of synchronization primitives. The solution presented in this section assumes that the pool consists of n buffers, each capable of holding one item.



The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer is as follows:

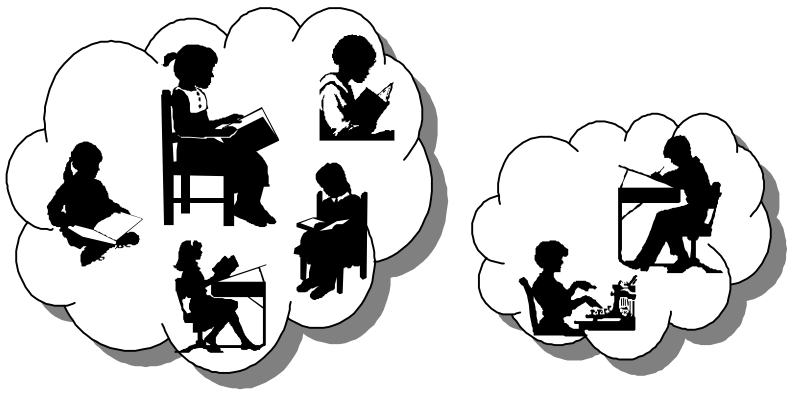
```
do {
   ...
   produce an item in nextp
   ...
   wait(empty);
   wait(mutex);
   ...
   add nextp to buffer
   ...
   signal(mutex);
   signal(full);
} while(1);
```

And that for the consumer is as follows:

```
do {
   wait(full);
   wait(mutex);
   ...
   remove an item from
   buffer to nextc
   ...
   signal(mutex);
   signal(empty);
   ...
   consume the item in nextc
   ...
} while(1);
```

Note the symmetry between the producer and the consumer process. This code can be interpreted as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

Readers Writers Problem



A data object (such as a file or a record) is to be shared among several concurrent processes. Some of these processes, called **readers**, may want only to read the content of the shared object whereas others, called **writers**, may want to update (that is to read and write) the shared object. Obviously, if two readers access the data simultaneously, no adverse effects will result. However, if a writer and some other process (whether a writer or some readers) access the shared object simultaneously, chaos may ensue.

To ensure these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The **second readers-writers problem** requires that once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we discuss a solution to the first readers-writers problem. In the solution to the first readers-writers problem, processes share the following data structures.

```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both the reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the reader processes update the readcount variable. The readcount variable keeps track of how many processes are currently reading the object. The wrt semaphore is used to ensure mutual exclusion for writers or a writer and readers. This semaphore is also used by the first and last readers to block entry of a writer into its critical section and to allow open access to the wrt semaphore, respectively. It is not used by readers who enter or exit, while at least one reader is in its critical sections.

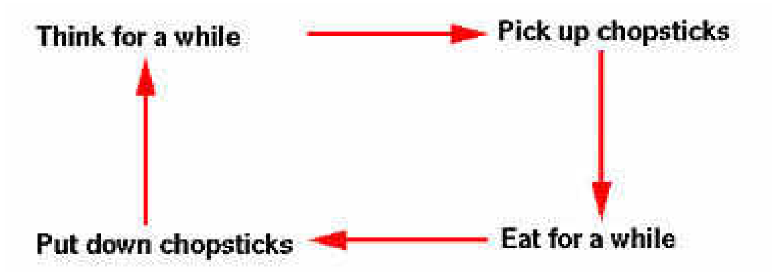
The codes for reader and writer processes are shown below:

```
wait(mutex);
  readcount++;
  if(readcount == 1)
     wait(wrt);
signal(mutex);
   ...
   reading is performed
   ...
wait(mutex);
  readcount--;
  if(readcount == 0)
     signal(wrt);
signal(mutex);
```

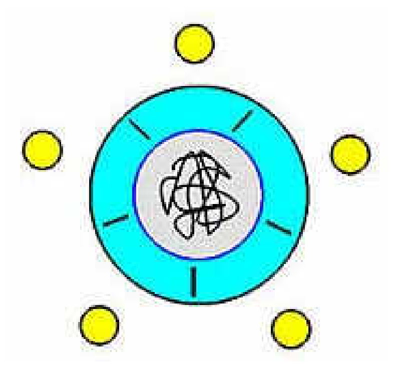
```
wait(wrt);
   ...
   writing is performed
   ...
signal(wrt);
```

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex. Also observe that when a writer executes signal(wrt) we may resume the execution of either the waiting readers or a single waiting writer; the selection is made by the CPU scheduler.

Dining Philosophers Problem
Consider five philosophers who spend their lives thinking and eating, as shown in the following diagram.



The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.



When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of her neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining philosophers problem is considered to be a classic synchronization problem because it is an example of a large class of concurrency control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tires to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus the shared data are:

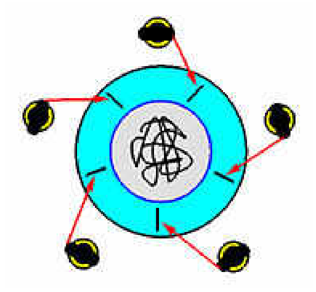
```
semaphore chopstick[5];
```

All the chopsticks are initialized to 1. The structure of philosopher i is as follows:

```
do {
   wait(chopstick[i];
   wait(chopstick[(i+1)%5]);
   ...
   eat
   ...
   signal(chopstick[i]);
   signal(chopstick[(i+1)%5]);
   ...
   think
   ...
}
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock.

Suppose that all five gets hungry at the same time and pick up their left chopsticks as shown in the following figure. In this case, all chopsticks are locked and none of the philosophers can successfully lock her right chopstick. As a result, we have a circular waiting (i.e., every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.



There are several possible good solutions of the problem. We will discuss these in the next lecture.

Operating Systems
Lecture No. 25

Reading Material

- Chapter 7 of the textbook
- Lecture 25 on Virtual TV

Summary

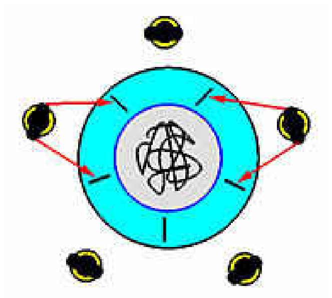
- Dining philosophers problem
- High-level synchronization constructs
- Critical region
- Monitor

Dining Philosophers Problem

Several possibilities that remedy the deadlock situation discussed in the last lecture are listed. Each results in a good solution for the problem.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Removing the possibility of deadlock does not ensure that starvation does not occur. Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a starvation. Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!



High-level Synchronization Constructs

We discussed the problems of deadlock, starvation, and violation of mutual exclusion caused by the poor use of semaphores in lecture 23. We now discuss some high-level synchronization constructs that help solve some of these problems.

Critical regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect usage can still result in timing errors that are difficult to detect, since these errors occur only if some particular execution takes place, and these sequences do not always happen.

To illustrate how, let us review the solution to the critical section problem using semaphores. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

To deal with the type of errors we outlined above and in lecture 23, a number of high-level constructs have been introduced. In this section we describe one fundamental high-level synchronization construct—the **critical region**. We assume that a process consists of some local data, and a sequential program that can operate on the data. Only the sequential program code that is encapsulated within the same process can access the local data. That is, one process cannot directly access the local data of another process. Processes can however share global data.

The critical region high-level synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as:

```
v:shared T;
```

The variable v can be accessed only inside a region statement of the following form:

```
region v when B do S;
```

This construct means that, while statement S is being executed, no other process can access the variable v. The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v. Thus if the two statements,

```
region v when(true) S1;
region v when(true) S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "S1 followed by S2" or "S2 followed by S1".

The critical region construct can be effectively used to solve several certain general synchronization problems. We now show use of the critical region construct to solve the bounded buffer problem. Here is the declaration of buffer:

```
struct buffer {
   item pool[n];
   int count,in,out;
};
```

The producer process inserts a new item (stored in nextp) into the shared buffer by executing

```
region buffer when(count < n) {
   pool[in] = nextp;
   in = (in+1)%n;
   count++;
}
```

The consumer process removes an item from the shared buffer and puts it in nextc by executing

```
region buffer when(count > 0) {
   nextc = pool[out];
   out = (out+1)%n;
   count--;
}
```

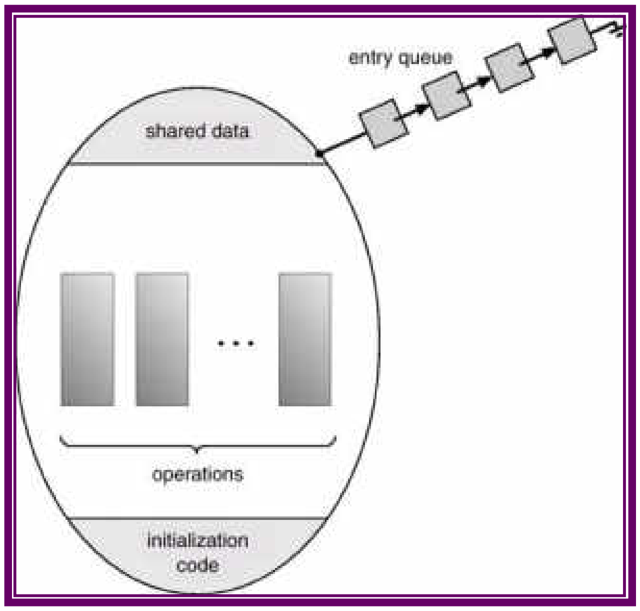
Monitors

Another high-level synchronization construct is the monitor type. A **monitor** is characterized by local data and a set of programmer-defined operators that can be used to access this data; local data be accessed only through these operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. Normal scoping rules apply to parameters of a function and to its local variables. The syntax of the monitor is as follows:

```
monitor monitor_name
{
   shared variable declarations

   procedure body P1(..) { ...}
   procedure body P1(..) { ...}
   ...
   procedure body P1(..) { ...}
   {
      initialization code
   }
}
```

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization construct explicitly. While one process is active within a monitor, other processes trying to access a monitor wait outside the monitor. The following diagram shows the big picture of a monitor.



However, the monitor construct as defined so far is not powerful enough to model some synchronization schemes. For this purpose we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct** (also called **condition variable**). A programmer who needs to write her own tailor made synchronization scheme can define one or more variables of type condition.

```
condition x,y;
```

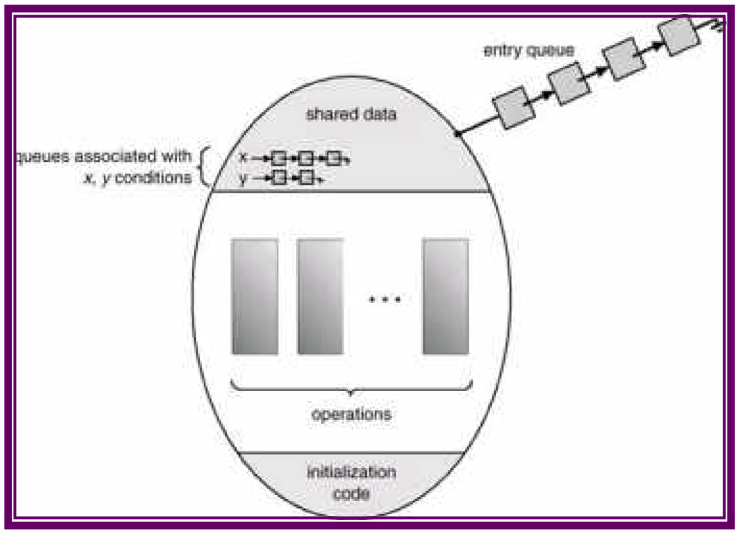
The only operations that can be invoked on a condition variable are wait and signal. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes.

```
x.signal();
```

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation were never executed. This is unlike the signal operation on a semaphore, where a signal operation always increments value of the semaphore by one. Monitors with condition variables can solve more synchronization problems that monitors alone. Still only one process can be active within a monitor but many processes may be waiting for a condition variable within a monitor, as shown in the following diagram.



In the next lecture we will discuss a monitor-based solution for the dining philosophers problem.

Operating Systems
Lecture No. 26

Reading Material

- Chapters 7 and 8 of the textbook
- Lecture 26 on Virtual TV

Summary

- Monitor-based solution of the dining philosophers problem
- The deadlock problem
- Deadlock characterization
- Deadlock handling
- Deadlock prevention

Monitor-based Solution for the Dining Philosophers Problem

Let us illustrate these concepts by presenting a deadlock free solution to the dining philosophers problem. Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution we need to distinguish among three states in which a philosopher may be. For this purpose we introduce the following data structure:

```
enum {thinking, hungry, eating} state[5];
```

Philosopher i can set the variable `state[i]=eating` only if her two neighbors are not eating: `(state[(i+4)%5]!=eating)` and `(state[(i+1)%5]!=eating)`.

We also need to declare five condition variables, one for each philosopher as follows. A philosopher uses her condition variable to delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

```
condition self[5];
```

We are now in a position to describe our monitor-based solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor dp; whose definition is as follows:

```
monitor dp
{
   enum {thinking,hungry,eating} state[5];
   condition self[5];

   void pickup(int i)
   {
      state[i]=hungry;
      test(i);
      if (state[i] != eating)
         self[i].wait();
   }
   void putdown(int i)
   {
      state[i]=thinking;
      test((i+4)%5);
      test((i+1)%5);
   }
   void test(int i)
   {
      if ((state[(i+4)%5]!=eating) &&
           (state[i]==hungry)&& state[(i+1)%5]!=eating)) {
         state[i]=eating;
         self[i].signal();
      }
   }
   void init()
   {
      for(int i=0;i<5;i++)
     state[i]=thinking;
   }
}
```

Each philosopher before starting to eat must invoke the pickup operation. This operation ensures that the philosopher gets to eat if none of its neighbors are eating. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown operation and may start to think. The putdown operation checks if a neighbor (right or left—in this order) of the leaving philosopher wants to eat. If a neighboring philosopher is hungry and neither of that philosopher's neighbors is eating, then the leaving philosopher signals it so that she could eat. In order to use this solution, a philosopher i must invoke the operations pickup and putdown in the following sequence:

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. You should think about this problem and satisfy yourself.

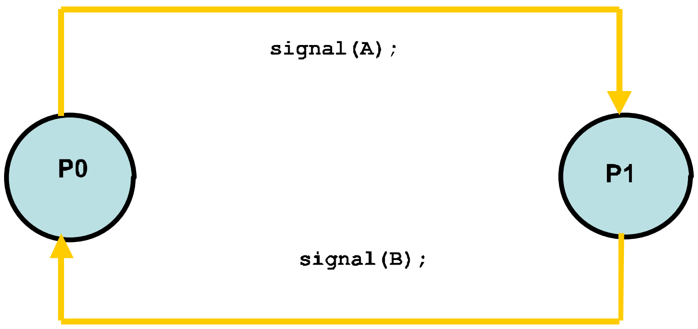
The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. Here's an example:

- System has 2 tape drives.
- P1 and P2 each hold one tape drive and each needs another one.

Another deadlock situation can occur when the poor use of semaphores, as discussed in lecture 23. We reproduce that situation here. Assume that two processes, P0 and P1, need to access two semaphores, A and B, before executing their critical sections. Semaphores are initialized to 1 each. The following code snippets show how a situation can arise where P0 holds semaphore A, P1 holds semaphore B, and both wait for the other semaphore—a typical deadlock situation as shown in the figure that follows the code.

P0	**P1**
wait (A);	wait(B);
wait (B);	wait(A);



In the first solution for the dining philosophers problem, if all philosophers become hungry at the same time, they will pick up the chopsticks on their right and wait for getting the chopsticks on their left. This causes a deadlock.

Yet another example of a deadlock situation can occur on a one-way bridge, as shown below. Traffic flows only in one direction, and each section of a bridge can be viewed as a resource. If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback). Several cars may have to be backed up if a deadlock occurs. Starvation is possible.