Recap 1

• Threads - multiple threads of execution in a single process
  – implementation
  – uses
  – difficulties

• IPC - race conditions

• Critical Regions
  – No two processes in their critical regions at once
  – No assumptions about speed or number of processors
  – No process outside its critical region can block another
  – No process waits forever to enter its critical region
Recap 2

- Strict Alteration
- Peterson’s Solution
- Test and Set
Semaphores 1

assume we have two operations $P$ and $V$ such that

$P$(semaphore)
critical section
$V$(semaphore)
non-critical section

satisfies the constraints we set out before

A **binary semaphore** abstracts the TAS or test and set constraint we had before
Semaphores 2

A binary semaphore has two states assume we will call them \textit{open} and \textit{closed}

then if we can make the following atomic, we win

\begin{verbatim}
P(semaphore) {
    while(semaphore==CLOSED);
    semaphore = CLOSED;
}
\end{verbatim}

\begin{verbatim}
V(semaphore) {
    semaphore = OPEN;
}
\end{verbatim}
Semaphores 3

• For any number of processes, this solution works
• Most modern operating systems provide a semaphore abstraction
• The typical way of implementing binary semaphores is with a queue- when you do a P if the semaphore is held by another process, you are put on a queue. Each V operation simply wakes the first process on the queue up if there is one
Semaphores 4

We can extend binary semaphores to what are called: **counting semaphores.** Counting semaphores take on non-negative integer values:

\[
\begin{align*}
\text{P(semaphore)} & \{ \\
& \text{while (semaphore} == 0); \\
& \text{semaphore--}; \\
\} \\
\text{V(semaphore)} & \{ \\
& \text{semaphore}++; \\
\}
\end{align*}
\]

The body of P is atomic.

Counting semaphores turn out to be surprisingly useful...
Producer/Consumer 1

Also called Bounded Buffer

• Two kinds of processes:
  • Producers put things in a buffer
  • Consumers remove things from buffer
• Producers must be aware of a full buffer condition
• Consumers must be aware of the case when there is nothing in the buffer (empty buffer condition)
Producer/Consumer 2

Assume we can have at most MAX items in the buffer:
sem1 = MAX;  // counting semaphore
sem2 = 0;     // counting semaphore
sem3 = OPEN;  // binary semaphore

Producer

while(TRUE) {
    P(sem1);
    P(sem3); add_item(); V(sem3);
    V(sem2);
}

Consumer

while(TRUE) {
    P(sem2);
    P(sem3); remove_item(); V(sem3);
    V(sem1);
}

The P(sem1) / V(sem2) gives us bounded alteration of processes. Note, if MAX == 1, we have strict alteration
Dining Philosophers

• Dijkstra proposed this *classic* problem
• 5 philosophers are sitting at a round table, they have plates of spaghetti in front of them. There are 5 forks on the table one between each plate. A philosopher needs two forks in order to eat. Each philosopher sits thinking for a while, gets hungry and tries to eat
• Obvious solution is to pickup the right fork and then pickup the left fork - results in **deadlock**
• Solution that uses a big lock around everything serializes things (bad)
• The code in the book is worth looking at
Readers and Writers 1

- Two kinds of processes
  - Readers
  - Writers
- We allow multiple readers at one time
- We do not allow multiple writers at one time
- We do not allow writers and readers at the same time
- This is a common problem in databases
Readers and Writers 2

Reader

P(lock1);
readct++;
if (readct == 1)
   P(dblock);
V(lock1);
read_db(); // read the db
P(lock1);
readct--;
if (readct = 0)
   V(dblock);
V(lock1);

Writer

P(dblock);
write_db();
V(dblock);
Readers and Writers 3

• This solution allows multiple readers, but we have the risk that the writer will never get to run
• One solution is to add locks such that once there is a writer waiting, no more readers are allowed in until the writer has written
• These sorts of tricks are surprisingly hard to get right - dead locks and race conditions abound
Scheduling 1

- Scheduling the processor or scheduling which job should run next
- Schedulers should try to meet the following objectives
  - Fairness - each process gets a fair amount of the processor
  - Efficiency - maximize the amount of usage the CPU gets
  - Response Time - minimize the response times for interactive users
  - Turnaround - minimize time it takes to finish processes
  - Throughput - maximize the number of processes run per unit time
  - Degrade gracefully and predictably under high load conditions
    - in other words, we need to be able to predict how much hardware we need - interestingly enough, Linux is much worse than NT for predictability under high load conditions
  - Repeatable
  - Doesn’t allow cheating
- These are not all achievable at the same time
Scheduling 2

• Preemptive scheduling is when the processor can interrupt one process to schedule another
• Non-preemptive scheduling is what it sounds like
• Scheduling algorithms we will talk about
  – FCFS
  – Round Robin
  – Round Robin with penalties
  – Processor Sharing
  – Shortest Job First
  – Preemptive Shortest Job First
  – Priority Scheduling
  – Priority with multiple queues
  – Other multi queue schemes
  – Various Real-Time scheduling algorithms
FCFS - First Come First Served

- Also called FIFO (first in first out)
- Keep a queue of processes, run the one on the queue until it is done, and then run the next one
- Non preemptive
- The simplest scheduling policy - early systems used this
Round Robin

• Preemptive scheduling policy
• Important question is how long each process gets to run for this is called the **quantum (q)**

How it works
  - a running process gets q time units to run.
  - when the timer goes off, the process gets preempted
  - the process gets moved to the ready queue
  - the next process in the ready queue gets to run
  - as q gets large, the Round Robin looks like FCFS
  - as q gets very small, we get processor sharing

• There are many issues involved in choosing how big or small to make q
  - small q makes the system responsive but wastes resources
  - large q makes the system more efficient but less responsive
Round Robin with Penalties

- Like Round Robin, but we can reduce the quantum $q$ of a process for various reasons including using all of its CPU time consistently
- We can also increase the number of occurrences of a process in the list - this would give the process multiple chances to run (i.e. increase the amount of CPU it got)
Processor Sharing

- Each of $N$ processes thinks it has all of a machine $1/N$ as fast as the actual machine.
- Actually implemented by the CDC 6600.
- Imagine RR scheduling with a very short time quantum.
Shortest Job First

- Given a list of processes (actually jobs) run the one first that will take the least time
- Non preemptive
- With no i/o switching, SJF has the shortest average waiting time
- Of course, predicting the time the job will take to run can be hard...
Preemptive SJF

- When a new job enters the run queue, if it has a shorter time than the remaining time of the job on the run queue, then we run the new job.
- Obviously, we can have starvation with this.
- One modification is to allow the jobs, as they wait, to decrease their predicted run times. This modification removes the possibility of starvation. This modification of the priority of the job as it waits is called **aging**.
Priority Scheduling

• Assign priorities to processes based on some scheme
• We have a list of processes, and we find the process with the highest priority and run it
• We can penalize or help processes based on things like
  – has been waiting a long time (aging)
  – has been using lots of CPU time
  – is interactive (or not interactive)
• This is a very popular scheduling algorithm
• The risk is that the computation of the priority may get expensive, and the system may spend a great deal of time computing which the best process to run next is
Priority Scheduling (multi queue)

- We can have multiple priority levels with a ready and a run queue associated with each level.
- A process is put in the appropriate queue based on its priority.
- The scheduling done in the queues themselves can be round robin.
- We risk starvation if the high priority queues use all of the time.
  - One solution is to use aging to allow the lower priority queues to run now and then.
Kernel Mode Priorities

<table>
<thead>
<tr>
<th>Not Interruptible</th>
<th>Interruptible</th>
<th>User Priorities</th>
</tr>
</thead>
<tbody>
<tr>
<td>swapper</td>
<td>waiting for TTY input</td>
<td>user level 0</td>
</tr>
<tr>
<td>waiting for disk i/o</td>
<td>waiting for TTY output</td>
<td>user level 1</td>
</tr>
<tr>
<td>waiting for buffer</td>
<td>waiting for child exit</td>
<td>user level n</td>
</tr>
</tbody>
</table>
Other Multi Queue

- Move jobs from queue to queue to separate batch from interactive processes
  - Run jobs in highest priority queue (might be RR)
  - If job uses all of its quantum, move it to a lower priority queue
  - If job doesn’t use all of its quantum, move it to a higher queue
    - these two try to separate batch from interactive
  - We can use different scheduling algorithms on different queues
- There are many variants of this basic idea
Real-Time Scheduling

- Real-Time falls into two categories
  - Hard real-time - a failure to meet a deadline causes the system to fail
  - Soft real-time - statistical deadlines need to be met
Real-Time Scheduling

- Look for guarantees i.e. a schedule that satisfies the constraints will be found
- Earliest deadline first is often the basis for the scheduling algorithm
- Priority based schemes are common
- Pre-computed schedules
Two Level Scheduling

- scheduling when the process may or may not be in memory
- requires a way for the scheduler to know whether the process has been swapped out
- to avoid starvation, the scheduler must have a way to have the process brought back into memory
- Tannenbaum assumes that a higher level scheduler decides when to bring a process back into memory
- We will deal with these issues when we talk about memory management
Scheduling Issues

- Threads
- Very large numbers of processes
- Degrading gracefully
- Scheduling very long running processes
- Scheduling and the swapper interfering
- Scheduling jobs and interactive processes at the same time
- Queuing theory to characterize the behavior of different scheduling algorithms