California State University, East Bay
Department of Mathematics and Computer Science

CS 6560

Operating Systems Design (Theory)

Instructor: Ted Billard
Operating Systems: Outline

I. Process Scheduling
II. Semaphores
III. Classical Problems
IV. Progress/Fairness
V. Deadlock/Banker’s Algorithm
VI. Memory Management
VII. Disk Scheduling
[I] Process Scheduling

summary

- process is the execution of a program
- creation, execution, deletion of processes
- OS schedules which process gets the CPU next
- many processes appear to run concurrently
- orderly (synchronized) access to shared data
- Interprocess Communication (IPC)
  - semaphores and shared data
  - messages
| **program** | • **program**: source code compiled to executable  
|            | • passive entity and resides on disk |
| **process** | • **process**: a program in execution  
|            | • active entity and the unit of work  
|            | • **data** for the variables used by the **instructions**  
|            | • also called a **job** or **task**  
|            | • long time to switch **heavyweight** processes |
| **thread**  | • short time to switch **lightweight** processes  
|            | • subprocesses inside a process |
| **CPU**     | • **Central Processing Unit** or **processor**  
<p>|            | • executes processes, performing the <strong>instructions</strong> |</p>
<table>
<thead>
<tr>
<th>Process: Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>running</strong></td>
</tr>
<tr>
<td>• one process is <em>running</em>, or executing, on the CPU</td>
</tr>
<tr>
<td><strong>concurrent</strong></td>
</tr>
<tr>
<td>• time-share CPU - appear to run simultaneously</td>
</tr>
<tr>
<td><strong>cooperating</strong></td>
</tr>
<tr>
<td>• interact and affect each other</td>
</tr>
<tr>
<td>• share data or resources</td>
</tr>
<tr>
<td>• <strong>synchronous</strong></td>
</tr>
<tr>
<td>• <strong>asynchronous</strong>: not cooperating (no interaction)</td>
</tr>
<tr>
<td><strong>state</strong></td>
</tr>
<tr>
<td>• current activity or condition (e.g. running)</td>
</tr>
<tr>
<td><strong>suspended</strong></td>
</tr>
<tr>
<td>• waiting for memory - then joins the <em>ready queue</em></td>
</tr>
<tr>
<td>State</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>
| **ready** | - waiting for access to the CPU  
              - not waiting for I/O or some other event  
              - in the **ready queue** - list of process IDs (PIDs)  
              - ready → running: **scheduled** for CPU  
              - running → ready: **interrupted** by scheduler  
              - running → **waiting** |
| **waiting** | - not ready to use the CPU  
                  - waiting for I/O or memory  
                  - waiting for a **semaphore** to be signaled  
                  - waiting for a **message** to be received |
| **device queue** | - waiting for **Input**  
                      - waiting for **Output** |
**Figure 4.1**

**Process: State Changes**

- **new** processes admitted to **ready** queue
- **ready** process is **scheduled** for CPU
- current running process returns to **ready**
- or **terminates**
- or **waits** for I/O or other event
Process Control Block (PCB)

- each process has its own PCB
- program counter (pc) points to the next instruction to be executed
- information is sufficient to stop/restart process (context switch)
  - store all registers, etc. in PCB₁
  - put PID₁ back in ready queue (or other queue)
  - remove PID₂ from ready queue, load registers based on PCB₂
  - start executing PID₂ at pc register
- each **process** (or **thread**) has a **pc**
- increments **sequentially** but branches for loops and conditionals
- CPU is switched to another process
- state (PCB) of the old process is saved
- saved state of the new process is loaded
Interrupts

- event that alters the sequence of instruction execution
- OS is interrupt driven:
  - sits quietly (no polling) until told there is something to do
- interrupt is generated by hardware or software

Steps:
- interrupts (usually) are disabled to prevent new ones
- OS gains control of CPU
- OS saves state of interrupted process (if user process: PCB)
- OS analyzes interrupt, passes control to interrupt handler routine
  - predefined number of routines
  - index into table (interrupt vector) that points to routines
- routine processes interrupt
- restore state of interrupted process (or some “next” process)
- interrupts (usually) are enabled to allow new ones
- interrupted process (or “next”) executes
Interrupts:

- **Software interrupts:**
  - program check interrupt: division by zero, bad memory location
  - system call to OS kernel (trap)
    - kernel is aware of process crossing its border
    - example: `read()`
    - kernel's device driver processes request
      - loads registers in device controller and starts controller
    - controller transfers data to buffer
    - when controller is finished: generates a hardware I/O interrupt

- **Hardware interrupts:**
  - I/O interrupt
    - I/O completed, CPU can restart user process or "next" process
  - external interrupt: expiration of quantum on clock
    - allows OS dispatcher to **context switch** to next process
### CPU Scheduling

**Summary**

- Only one process at a time is **running** on the CPU.
- A process gives up the CPU:
  - If it starts waiting for an **event**.
- Otherwise, other processes need **fair access**.
- OS **schedules** which **ready** process to run next.
- **Time slice** or **quantum** for each process.
- Scheduling algorithms:
  - Different goals.
  - Affect performance.

---

**[13]**
<table>
<thead>
<tr>
<th>Scheduling: Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>long-term scheduler</strong></td>
</tr>
<tr>
<td>- job scheduler</td>
</tr>
<tr>
<td>- which process on disk should be given memory?</td>
</tr>
<tr>
<td>- result: new process in <strong>ready queue</strong></td>
</tr>
<tr>
<td>- important in batch systems</td>
</tr>
<tr>
<td>- many processes in memory ⇒</td>
</tr>
<tr>
<td>high <strong>degree of multiprogramming</strong></td>
</tr>
<tr>
<td><strong>short-term scheduler</strong></td>
</tr>
<tr>
<td>- <strong>CPU</strong> scheduler</td>
</tr>
<tr>
<td>- which process in ready queue should be given <strong>CPU</strong></td>
</tr>
<tr>
<td>- result: new process on <strong>CPU</strong></td>
</tr>
<tr>
<td>Scheduling: Definitions</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td><strong>CPU-bound</strong></td>
</tr>
<tr>
<td><strong>I/O-bound</strong></td>
</tr>
</tbody>
</table>
| multilevel scheduling   | • classified into different groups  
|                         | • foreground (interactive) vs.  
|                         | • background (batch)  
|                         | • each group has its own ready queue |
| **utilization** | percentage of time that the CPU is busy.  
|                | if not busy, ready queue must be empty  
|                | - CPU actually executes NULL process  
|                | goal: keep the CPU busy |
| **throughput** | if busy, then work is being done  
|                | number of processes completed per second |
| **turnaround** | total time to complete a process  
|                | includes waiting in the ready queue  
|                | executing on the CPU  
|                | waiting for I/O  
<p>|                | goal: fast turnaround |</p>
<table>
<thead>
<tr>
<th><strong>[I] Performance: Definitions</strong></th>
<th>[17]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>response</strong></td>
<td></td>
</tr>
<tr>
<td>• time waiting in the <strong>ready queue</strong> and</td>
<td></td>
</tr>
<tr>
<td>• executing on CPU until some output produced</td>
<td></td>
</tr>
<tr>
<td>• average is across all output events</td>
<td></td>
</tr>
<tr>
<td>• goal: fast response time</td>
<td></td>
</tr>
<tr>
<td><strong>waiting</strong></td>
<td></td>
</tr>
<tr>
<td>• sum of periods spent waiting in <strong>ready queue</strong></td>
<td></td>
</tr>
<tr>
<td>• average is across all visits to ready queue</td>
<td></td>
</tr>
<tr>
<td>• goal: short waiting time</td>
<td></td>
</tr>
<tr>
<td><strong>important</strong></td>
<td></td>
</tr>
<tr>
<td>• scheduler has a direct effect on waiting time</td>
<td></td>
</tr>
<tr>
<td>• decides which process in queue gets to run next</td>
<td></td>
</tr>
<tr>
<td>• remaining processes must then wait longer</td>
<td></td>
</tr>
<tr>
<td>• OS cannot control code, amount of I/O, etc.</td>
<td></td>
</tr>
</tbody>
</table>
Performance: Summary

- **UTILIZATION:** CPU % busy
- **THROUGHPUT:** jobs/sec
- **WAITING:** sec/job
- **RESPONSE:** sec/job (usually in time-share systems)
- **TURNAROUND:** sec/job (usually in batch systems)

```
  READY       CPU       I/O
  WAITING (W)  
  RESPONSE (R)  
  TURNAROUND (R_{sys})
```
CPU burst

- cycle of CPU burst, I/O wait, CPU burst, ...
- program and data determine length of burst
- scheduler may interrupt a burst
- but does not affect the full length

```c
scanf n, a, b  /* I/O wait */
for (i=1; i<=n; i++) /* CPU burst */
    x = x + a*b;
printf x    /* I/O wait */
for (i=1; i<=n; i++) /* CPU burst */
    for (j=1; j<=n; j++)
        x = x + a*b;
printf x    /* I/O wait */
```
Scheduling: FCFS

- First-Come, First-Served is simplest scheduling algorithm
- ready queue is a **FIFO** queue: First-In, First-Out
- longest waiting process at the front (head) of queue
- new ready processes join the rear (tail)

**nonpreemptive**: executes until voluntarily gives up CPU
  - finished or waits for some event

problem:
  - **CPU-bound** process may require a long CPU burst
  - other processes, with very short CPU bursts, wait in queue
  - reduces CPU and I/O device utilization
  - it would be better if the shorter processes went first
[I] Scheduling: FCFS

- assume processes arrive in this order: \( P_1, P_2, P_3 \)
- **nonpreemptive** scheduling
- average waiting time: \( (0+24+27)/3 = 17 \) ms

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>24</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
</tr>
</tbody>
</table>

**Gantt chart**

\[
\begin{array}{c}
0 & P_1 & | & P_2 & P_3 \\
\hline
24 &  &  & 27 & 30 \\
\end{array}
\]

- assume processes arrive in this order: \( P_2, P_3, P_1 \)
- average waiting time: \( (6+0+3)/3 = 3 \) ms

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_2 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>24</td>
</tr>
</tbody>
</table>

**Gantt chart**

\[
\begin{array}{c}
0 & P_2 & | & P_3 & | & P_1 \\
\hline
3 & 6 & | & 30 & \\
\end{array}
\]

- **in general**, **FCFS** average waiting time is not minimal
- **in general**, better to process shortest jobs first
<table>
<thead>
<tr>
<th>I</th>
<th>Scheduling: Round Robin (RR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• similar to <strong>FCFS</strong>, but <strong>preemption</strong> to switch between processes</td>
<td></td>
</tr>
<tr>
<td>• <strong>time quantum</strong> (time slice) is a small unit of time (10 to 100 ms)</td>
<td></td>
</tr>
<tr>
<td>• process is executed on the CPU for at most one time quantum</td>
<td></td>
</tr>
<tr>
<td>• implemented by using the <strong>ready queue</strong> as a <strong>circular queue</strong></td>
<td></td>
</tr>
<tr>
<td>• <strong>head</strong> process gets the CPU</td>
<td></td>
</tr>
<tr>
<td>• uses less than a time quantum ⇒ gives up the CPU voluntarily</td>
<td></td>
</tr>
<tr>
<td>• uses full time quantum ⇒ <strong>timer</strong> will cause an <strong>interrupt</strong></td>
<td></td>
</tr>
<tr>
<td>• <strong>context switch</strong> will be executed</td>
<td></td>
</tr>
<tr>
<td>• process will be put at the <strong>tail</strong> of queue</td>
<td></td>
</tr>
</tbody>
</table>
[I] Scheduling: RR

- assume processes arrive in this order: \( P_1, P_2, P_3 \)
- preemptive scheduling
- time quantum: 4 ms
- \( P_1 \) uses a full time quantum; \( P_2, P_3 \) use only a part of a quantum
- \( P_1 \) waits 0+6=6; \( P_2 \) waits 4; \( P_3 \) waits 7
- average waiting time: \((6+4+7)/3=5.66\) ms

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>24</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
</tr>
</tbody>
</table>

Gantt chart

\[
\begin{array}{cccccccc}
 & P_1 & P_2 & P_3 & P_1 & P_1 & P_1 & P_1 & P_1 \\
0 & 4 & 7 & 10 & 14 & 18 & 22 & 26 & 30 \\
\end{array}
\]

- very large time quantum \( \Rightarrow \) RR = FCFS
- very small time quantum \( \Rightarrow \) context switch is too much overhead
- quantum \( \approx \) CPU burst \( \Rightarrow \) better turnaround
  - rule of thumb: 80% should finish burst in 1 quantum
I. Scheduling: Shortest-Job-First (SJF) [24]

- assume the next burst time of each process is known
- SJF selects process which has the shortest burst time
- optimal algorithm because it has the shortest average waiting time
- impossible to know in advance
- OS knows the past burst times - make a prediction using an average
- nonpreemptive
- or preemptive:
  - shortest-remaining-time-first
  - interrupts running process if a new process enters the queue
  - new process must have shorter burst than remaining time
[I] Scheduling: SJF

• assume all processes arrive at the same time: $P_1, P_2, P_3, P_4$
• nonpreemptive scheduling
• average waiting time: $(3+16+9+0)/4=7$ ms

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

Gantt chart

<table>
<thead>
<tr>
<th></th>
<th>$P_4$</th>
<th>$P_1$</th>
<th>$P_3$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

• SJF is optimal: shortest average waiting time
• but burst times are not known in advance
• next_predicted burst time by (weighted) average of past burst times
  • $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$
  • next_predict = $\alpha \cdot$ last_observed + $(1 - \alpha) \cdot$ last_predict
  • $\alpha = 0 \Rightarrow$ next_predict = initialized value (usually 0)
  • $\alpha = 1 \Rightarrow$ next_predict = last_observed
[I] SJF: Weighted Average Burst

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \cdots + (1 - \alpha)^{n+1} \tau_0 \]

\[ \alpha = 1 : \tau_{n+1} = t_n \]

\[ \alpha = 0 : \tau_{n+1} = \tau_0 \]

\[ \alpha = 1/2 : \text{recent and past history the same} \]

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burst ((t_i))</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Guess ((\tau_i))</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

\[ \tau_1 = \frac{1}{2} t_0 + \frac{1}{2} \tau_0 = \frac{1}{2} 6 + \frac{1}{2} 10 = 8 \]
Scheduling: SJF

- assume processes arrive at 1 ms intervals: $P_1, P_2, P_3, P_4$
- preemptive scheduling: shortest-remaining-time-first
  - $P_1$ waits $0+(10-1)=9$; $P_2$ waits $1-1=0$
  - $P_3$ waits $17-2=15$; $P_4$ waits $5-3=2$
- average waiting time: $(9+0+15+2)/4=6.5$ ms

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Gantt chart

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_4$</th>
<th>$P_1$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

- nonpreemptive SJF: 7.75 ms
[I] Scheduling: Priority (PRIO)

- assume a priority is associated with each process
- select highest priority process from the ready queue
- let $\tau$ be the (predicted) next CPU burst of a process
- **SJF** is a special case of priority scheduling
  - assume: high numbers $\Rightarrow$ high priority
    - then priority is $1/\tau$
  - assume: low numbers $\Rightarrow$ high priority
    - then priority is $\tau$
- equal-priority processes are scheduled in **FCFS** order
- PRIO can be **preemptive** or **nonpreemptive**
- priorities can be defined **internally**
  - memory requirements, number of open files, burst times
- priorities can be defined **externally**
  - user, department, company
[I] Scheduling: PRIO

- assume all processes arrive at the same time: $P_1, P_2, P_3, P_4, P_5$
- nonpreemptive scheduling
- high priority: low number
- some OS use a high number!!! See VOS.
- average waiting time is: $(6+0+16+18+1)/5=8.2\, \text{ms}$

<table>
<thead>
<tr>
<th>PID</th>
<th>Burst</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Gantt chart**

```
  P_2 | P_5     | P_1     | P_3 | P_4
  0   1    6         16  18  19
```

- indefinite blocking (starvation): low priority process never runs
- aging: low priorities increase with waiting time, will eventually run
[II] Semaphores

- semaphores and classical problems
- interprocess communication (IPC)
- cooperating processes:
  - synchronized (orderly) access to shared globals
- process control:
  - mechanism to prevent execution until a certain event occurs
  - send of a message
  - wakeup of a sleeping process
  - signal of a semaphore
    - wait and signal guarantee only one process at a time executes a critical section of code
  - protects the shared access to global variables
program code: x=0; x++; print x;
   output: 1

- 3 processes execute same code with different output:

<table>
<thead>
<tr>
<th>PID 1</th>
<th>PID 2</th>
<th>PID 3</th>
<th>x: (global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0;</td>
<td>x=0;</td>
<td>x=0;</td>
<td>0</td>
</tr>
<tr>
<td>x++;</td>
<td>x++;</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>x=0;</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>printf x;</td>
<td>printf x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x++;</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>printf x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 &lt;=== output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x=0;</td>
<td>1 &lt;=== output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>printf x;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x++;</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>printf x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 &lt;=== output</td>
<td></td>
</tr>
</tbody>
</table>

- 2 processes execute i++ and time-slice after memory fetch:

<table>
<thead>
<tr>
<th>PID 1</th>
<th>PID 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch i = 0 from memory</td>
<td>fetch, load, increment, store 1 in memory, slice</td>
</tr>
<tr>
<td>load on register</td>
<td></td>
</tr>
<tr>
<td>time-slice</td>
<td></td>
</tr>
<tr>
<td>increment i</td>
<td></td>
</tr>
<tr>
<td>store 1 in memory</td>
<td></td>
</tr>
</tbody>
</table>
[II] Critical Section: Attempted Solution

**Shared Variable:**

```c
int v=0; /* v==0 => critical section OPEN; v==1 => critical section CLOSED */
```

**Code:** At \( p_i \in \{1, ..., n\} \):

```c
while (1) {
    /* trying region */
    while (v == 1) /* do nothing */ ;

    v = 1;
    /* critical section (region) */
    x = 0;
    x++;
    printf x;

    v = 0;
    /* remainder region */
}
```

- what is wrong? what if critical section is: `dial_phone("555-1212")`?
- how can it be fixed?  
  - is this spinlock good or bad?
[II] Critical Section: General Problem

- \( n \) processes each with segment of code called **critical section**
- one process changes common (shared) variables, writes to a file.
- no other process is allowed to execute in its critical section
- execution of critical sections is **mutually exclusive** in time
- one solution: **semaphore**
  - lets **one** process into the critical section
  - puts other processes in a **semaphore queue** (no **busy waiting**)
  - process is finished: **head** of FIFO queue enters section
[II] Semaphore: wait and signal

(1) Train 1 Arrives

(2) Train 2 Arrives

(3) Train 1 Leaves

(4) Train 2 Leaves
[II] Semaphore: wait and signal

- critical section: general solution

```c
while(1) {
    pm_wait(sem);
    /* critical section or region */
    pm_signal(sem);
}
```

- critical section: example solution

```c
while(1) {
    pm_wait(sem);
    x=0;
    x++;
    printf x;
    pm_signal(sem);
}
```
[II] Semaphore: wait and signal

• semaphore has **count** and FIFO **queue** of waiting processes

  ```
  struct {
    int count=1;
    FIFO queue;
  } semaphore;
  ```

• **count** ≥ 0 ⇒ queue is **empty**
• **count** of negative $n$ ⇒ queue has $n$ **waiting** processes

  ```
  wait(semaphore) : if (--semaphore.count<0){
    put_at_tail(pid,semaphore.queue);
    suspend(pid) }
  ```

  ```
  signal(semaphore) : if (semaphore.count++<0){
    pid=get_at_head(semaphore.queue);
    ready(pid) }
  ```
## Semaphore: Initialization

- usually first process to `wait` is allowed access to **critical section**
- next process to `wait` is placed on the **semaphore queue**
- until the first process is finished (**signal**) 
- to guarantee this **first** access, initialize either:
  - by setting `count=1`:
    ```c
    sem = pm_seminit(1);
    ```
  - by setting `count=0` and then **signal**:
    ```c
    sem = pm_seminit(0);
    pm_signal(sem);
    ```
[II] Semaphore: Count

<table>
<thead>
<tr>
<th>PID 1</th>
<th>PID 2</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem=pm_seminit(1);</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>pm_wait(sem);</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>(critical)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>v</td>
<td>pm_wait(sem);</td>
<td></td>
</tr>
<tr>
<td>pm_signal(sem);</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>pm_wait(sem);</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>(critical)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>pm_wait(sem);</td>
<td></td>
</tr>
<tr>
<td>pm_signal(sem);</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>(critical)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>pm_wait(sem);</td>
<td></td>
</tr>
<tr>
<td>pm_signal(sem);</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Critical Section: Deadlock

- deadlock:
  - all processes are waiting indefinitely for some event to occur
  - that can only be caused by one of the waiting processes
  - “mutual waiting”
  - none of the processes make progress

Kansas legislature:

When two trains approach each other at a crossing,
both shall come to a full stop
and neither shall start up again until the other has gone.
[II] Semaphores: Deadlock

- process 1 **waits** for semaphore S and Q:

  ```
  pm_wait(S);
  pm_wait(Q);
  /* critical section */
  pm_signal(S);
  pm_signal(Q);
  ```

- process 2 **waits** for semaphore Q and S:

  ```
  pm_wait(Q);
  pm_wait(S);
  /* critical section */
  pm_signal(Q);
  pm_signal(S);
  ```

- both processes are **deadlocked** as each are **waiting** for the other
### Classical Problems: Introduction

- Classical problems are simple but represent complex, real problems
- Show important features of real problem/solution
- Examples from OS and real-time applications
- Main points:
  - Mutual exclusion to critical section code
  - Synchronized access to shared resources and data
[III] Dining Philosophers

**Problem**
- 2 philosophers
- 2 chopsticks (1 pair)

**Questions**
- How can both philosophers eat?
- How to share resources?

**Answer**
- TAKE TURNS USING RESOURCES
<table>
<thead>
<tr>
<th>OK</th>
<th>BAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philosopher 1 uses the 2 chopsticks (1 pair) to eat. Then Philosopher 2 uses the 2 chopsticks (1 pair) to eat. Both make <strong>progress</strong> and the access to the <strong>resources</strong> (chopsticks) is <strong>synchronized</strong>.</td>
<td>Philosopher 1 gets one (1) chopstick. Philosopher 2 gets the other (1) chopstick. Neither can eat. Both are <strong>deadlocked</strong>.</td>
</tr>
</tbody>
</table>
[III] Dining Philosophers

- each chopstick is a semaphore
- a philosopher waits for access to 2 chopsticks
- a philosopher signals when finished eating

```c
while(1) {
    /* philosopher thinks */
    pm_wait(chopstick1);
    pm_wait(chopstick2);
    /* philosopher eats */
    pm_signal(chopstick1);
    pm_signal(chopstick2);
}
```
Shared Variables:

1. \textit{FORK}: semaphore array \([0..n-1]\), initially all available

Code: At \(p_i \in \{0, ..., n - 1\}:

\[
\text{do forever}
\]

\[
\text{if even}(i) \text{ then }
\]

\[
\text{wait}(\text{FORK}_{i+1}) \quad /* \text{left fork} */
\]
\[
\text{wait}(\text{FORK}_i) \quad /* \text{right fork} */
\]

\[
\text{if odd}(i) \text{ then }
\]

\[
\text{wait}(\text{FORK}_i) \quad /* \text{right fork} */
\]
\[
\text{wait}(\text{FORK}_{i+1}) \quad /* \text{left fork} */
\]

\[
/* \text{Critical region} */
\]
\[
\text{signal}(\text{FORK}_i)
\]
\[
\text{signal}(\text{FORK}_{i+1})
\]
[III] Producer/Consumer

**Producer:**

```
data ⇒ buffer ⇒ data
```

**Consumer:**

---

### Questions

- What if producer is faster than consumer?
- What if buffer overflows?
- What if consumer is faster than producer?
- What if buffer is empty?

### Answer

- PRODUCER WAITS FOR NOT FULL
- CONSUMER WAITS FOR NOT EMPTY
Process 1 is a **producer** of data. Process 2 is a **consumer** of data.

Rule 1: the producer puts data in the **buffer**.

Rule 2: the buffer holds only 5 numbers.

Rule 3: the consumer gets data from the buffer.

Rule 4: the producer waits for the buffer to be **not full**.

Rule 5: the consumer waits for the buffer to be **not empty**.

**OK**  
Producer puts data into buffer; consumer gets data and writes it.

**BAD**  
The producer fills the buffer with 5 numbers, then **waits** for the consumer to **signal** that the buffer is **not full**. The consumer reads from the buffer, then **waits** for the producer to **signal** that the buffer is **not empty**. Both are **deadlocked**.
III Producer/Consumer

- producer *waits* for the buffer to be *not full*
- producer *waits* for access to the buffer (*mutual exclusion*)
- producer *signals* after access to the buffer (*mutual exclusion*)
- producer *signals* that the buffer is *not empty*

```c
while(1) {
    pm_wait(not_full);
    pm_wait(sem_buffer);
    /* put data into buffer */
    pm_signal(sem_buffer);
    pm_signal(not_empty);
}
```
[III] Readers/Writer

Problem:
- Many readers OK: $\text{FILE} \rightarrow \text{data}$
- One writer OK: $\text{FILE} \leftarrow \text{data}$
- Read/write NOT OK: data $\rightarrow \text{FILE} \rightarrow \text{data}$

Questions:
- What if readers are reading data from file?
- What if writer is writing data to file?

Answer:
- READERS WAIT FOR NO WRITERS
- WRITER WAITS FOR SOLE ACCESS
[III] Readers/Writer

Process 1 and 2 read data from a file.
Process 3 writes data to the file.

Rule 1: if a process reads data, another process can read data.
Rule 2: if a process writes data, another process cannot read data.

OK

Readers 1 and 2 read from the file and Writer 3 waits. Then, Writer 3 writes to the file and Readers 1 and 2 wait. All make progress and the shared access to the file is synchronized. The readers see a consistent file.

BAD

Writer 3 writes to the file before Readers 1 and 2 are finished reading from the file. The readers do not see a consistent file. Writer 3 needs to use a semaphore.
• reader **waits** for writer to finish writing
• reader **signals** to writer when all readers are finished reading

```c
while(1) {
    pm_wait(sem_readcount);
    readcount++;
    if (readcount==1) pm_wait(sem_wrt);
    pm_signal(sem_readcount);
    ...
    pm_wait(sem_readcount);
    readcount--;
    if (readcount==0) pm_signal(sem_wrt);
    pm_signal(sem_readcount);
}
```

• **writer waits** for access to the file
• **writer signals** after access to the file

```c
while(1) {
    pm_wait(sem_wrt);
    /* write to file */
    pm_signal(sem_wrt);
}
```
### Gantt: Readers/Writer

Processes: R1 (Reader 1), R2, W1 (Writer 1), W2  
Data Structures: RC: ReadCount, SR: SemReadCount, SRQ: SR Queue, SW: SemWrite, SWQ: SemWrite Queue, RDQ: Ready Queue

Scheduling: arbitrary but sometimes only one process to run  
R:W(S)*: Process R blocks on Semaphore S because count of S is negative

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R2→SRQ</td>
<td>W1→SWQ</td>
<td>W2→SWQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-1</td>
<td></td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2→RDQ</td>
<td>W1→RDQ</td>
<td>W2→RDQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A mutual exclusion problem must be considered for three attributes:

- **mutual exclusion:**
  - guarantee only one process is executing its critical section
  - also called “safety”

- **progress:**
  - guarantee that at least one process makes steps
  - some work is being completed
  - also called “liveness”, opposite of “deadlock”

- **fairness:**
  - guarantee that all processes make progress
  - opposite of “starvation”
  - “bounded-waiting”, unfairness is “indefinite postponement”
  - unfairness is asymmetrical
  - one process works at expense of another
[IV] Progress/Fairness

DEFINITION/PROOF TECHNIQUE

MUTEX:
at most 1 proc in C.S.
can’t prove not mutex

NOT MUTEX:
at least 2 procs in C.S.
SLICE before locking

PROGRESS:
at least 1 proc does work, liveness
can’t prove not progress
[IV] Progress/Fairness

NOT PROGRESS:
0 procs do work, deadlock, but not starvation
all SPINNING (or BLOCKED on semaphores)

FAIRNESS:
all procs do work
can’t prove not fairness

NOT FAIRNESS:
1 proc does work at expense of another,
starvation, asymmetrical relationship
one proc releases but RETAKES lock,
other proc just misses doing work
Of course, it’s impossible to enumerate every possible proof of NOT MUTEX. So you really don’t prove that it has MUTEX. But you always try NOT MUTEX first and, if you can’t find such a proof and you feel more confident about the code, you very carefully say that it does guarantee MUTEX.
[IV] Progress/Fairness

This is not a proof technique of unfairness: I pick a priority scheduler, always run P0, but never run P1. P1 starves. Well, what happens if you are given some different code? Are you going to repeat the same proof? Then you are given some more code. You repeat the same argument? You’re not analyzing the code you’re given - and that’s the main point. You have to give P1 some opportunities to run on the CPU. But you can choose those opportunities (devil’s advocate), and choose these such that P0 has taken the lock back, and P1 just misses everytime.
[IV] Progress/Fairness

Silberschatz is much more specific: if a process is out of its critical section, it should not be able to block others from getting into the critical section. For example, consider an algorithm that allocates the critical section STRICTLY by alternating turns. If process B does not take its turn - because it is busy doing other work or because it has crashed - then process A cannot get into its critical section even though it is available. Note that this could also be considered some form of “deadlock” and the system does not exhibit “liveness”. Silberschatz says the selection cannot be postponed indefinitely. This phrase should be applied to the concept of “fairness”, i.e. we don’t want one particular process to be postponed indefinitely from getting into the critical section. So I would stay away from this phrase.

Bottom Line: Progress means that at least one process is getting work done.
[IV] Progress/Fairness

IMPORTANT: If two processes are deadlocked, i.e., no progress, this is NOT a proof of unfairness (starvation). Admittedly, they are not getting any work done, but unfairness results from ASYMMETRY, i.e., one process is getting work done at the expense of another process not getting work done.
[IV] Progress/Fairness

To prove NO with respect to safety, progress or fairness, you need to make a counter example, i.e., a proof by contradiction. And if you can prove, say NO to safety, then that has no bearing on progress or fairness. That is, all three are independent and there are no rules of thumb to help correlate the answers.

To prove YES is more difficult. You can’t try every possible counter example. But if you try hard to answer NO but can’t find a proof, then you carefully answer YES, after examining the logic of the code.
Deadlock

- finite number of **resources** distributed to competing processes
  - physical: memory space, CPU cycles, registers, I/O devices
  - logical: files, semaphores
- process **requests, uses, releases** resource
- do not want **deadlock**: set of procs waiting for release from one (or more) members of set
- single **instance**: must have CPU2 (only will do)
- **Avoidance** will be able to use graph theory
- multiple **instances**: tapeDrive1..tapeDriveN (any will do)
- **Avoidance** will not be able to use graph theory, must use **Banker’s**
[V] Overview of Methods

Deadlock

Prevention
- Single Instance Resource
  - Resource Allocation Graph

Avoidance
- Multiple Instance Resource
  - Banker's Alg

Detection
- Single Instance Resource
  - Wait-for Graph

Recovery
- Multiple Instance Resource
  - Time-varying Banker's Alg
- Process Preemption
- Resource Preemption
[V] Methods for Handling Deadlock

- never let deadlock occur
  - prevention: break ONE of 4 necessary conditions
  - avoidance: resources give advance notice of maximum use
    - single (graph) vs. multiple (Banker’s) instances
- let deadlock occur and do something about it
  - detection: search for problems periodically
    - single (graph) vs. multiple (Banker’s) instances
  - recovery: preempt processes or resources
- don’t worry about it (UNIX and other OS)
  - cheap: just reboot (it happens rarely)
Prevention: 4 Necessary Conditions

- mutual exclusion: at least one nonshareable resource
- hold and wait: at least one proc holds resource, waits for other
- no preemption: resources can only be voluntarily released
- circular wait: \( P_0 \) waiting for \( P_1 \) ... waiting for \( P_0 \)
  - circular wait \( \Rightarrow \) hold and wait
- if you can break any one of these conditions \( \Rightarrow \) no deadlock
[V] Break Necessary Condition

- break **mutual exclusion**:
  - read-only files are shareable
  - but some resources are **intrinsically** nonshareable (printers)
- break **hold and wait**:
  - request all resources in advance
  - request (tape, disk, printer)
  - release all resources before requesting new batch
    - request (tape, disk), release (tape, disk), request (disk, printer)
- disadvantages: low resource utilization, **starvation**
[V] Break Necessary Condition

- **break no preemption:**
  - process 1 requests resources already allocated to process 2:
    - process 1 forfeits its current resources
    - if process 2 is waiting for other resources: process 2 forfeits
  - used for resources whose state is easily saved/restored
    - CPU registers and memory space
    - but not printers or tape drives
- **break circular wait:**
  - order all resources by unique numbers (tape drives, etc.)
  - processes request resources in increasing order
[V] Detection

- single instance resource types:
  - periodically make a wait for graph
    (which processes are waiting for which resources)
  - check for cycles \(n^2\)
- multiple instance resource types:
  - use a time-varying banker’s algorithm
- how often should detection algorithm be run?
  - how often is deadlock likely?
  - how many processes will be affected?
Recovery

- **process termination**
  - abort all deadlocked processes
  - abort one process at a time until cycle eliminated
    - run detection algorithm each time
    - which process to abort next?
      - priority? CPU usage? how many resources?
- **resource preemption**
  - take resources from some processes and give them to others
  - **select victim**: cheapest cost?
  - **rollback**: to safe state and restart
  - **starvation**: deadlock could occur again, process restarts
    - include number of rollbacks in cost
Avoidance

- processes give advance notice about maximum usage of resources
- processes make actual requests when they need a resource
- avoidance algorithm: allocate request only if it yields a safe state
  
a safe sequence (SS) of processes exists
  such that each process can still get their maximum in sequence
- conceptually the processes could be run in this order
- but the OS does not schedule to run in this order
- OS uses ordinary time-slicing, etc.
- but if the worst-case arises, only N-1 processes will block
- there will be at least one process able to run with desired resources
- releases enough resources for at least another to complete, etc.
- this is what the definition of SS implies
Safe vs. Unsafe

- safe sequence (SS) implies safe state
- safe state implies impossible to deadlock
- Avoidance “avoids” unsafe states
- unsafe state may or may not be deadlock
- Avoidance just doesn’t take a chance
- can lead to lower resource utilization
- process P1 blocked on resource R1
- R1 assigned to P2
- note that R3 has multiple instances (two)
- no cycle $\Rightarrow$ no deadlock
- deadlock $\Rightarrow$ cycle
- cycle $\Rightarrow$ maybe deadlock
- single instance $\land$ cycle $\Rightarrow$ deadlock
- multiple instances $\land$ cycle $\Rightarrow$ do not know, use Banker’s instead
Avoidance with Multiple Instances

DEADLOCK: can NOT unravel, deadly embrace

NO DEADLOCK: unravel by running P1, P2, P3 (i.e. SS)
[V] **Single vs. Multiple Instances**

- both examples have cycles of **assign/blocked request**
- multiple instances: cycles don’t tell us anything
- that’s why we need the Banker’s Algorithm
- but with Single Instances:
- cycle of **assign/blocked request** implies deadlock
- but avoidance does not want to get to this state
- need to stay safe
- can use **claim** edges, which resources **might** request
- stay safe: do not allow cycle of **assign/claim**
- never will have deadlock: cycle of **assign/blocked request**
- see next slide
**Avoidance with Single Instances**

SAFE: no cycle of *assign/claim*. SS: P1, P2

UNSAFE: cycle of *assign/claim*. No SS
Consider these graph edges (do not follow Silberschatz):

- **claim** edge: process P might need resource R in future
- arrow points towards resource
- Avoidance: must always specify potential requirements at start
- **assignment** edge: P requests R (has a claim first), assign it
- arrow points towards owner P
- **blocked request** edge: P requests unavailable R
- arrow points towards R, and “pointing” matters
- Avoidance: do not allow a cycle of assign/claim
- note a cycle of assign/claim does not mean deadlock
- prevents potential cycle of assign/blocked request (deadlock)
• Initial State A
• P1, P2 both claim R1, R2
• might need these resourses
• 2 SS: P1, P2 and P2, P1
• no cycles
• if necessary, they could finish
• same as Dining Philosophers

State B could follow State A
P1 requests R1
1 SS: P1, P2
no cycles
request granted, assigned
if necessary, P1 would finish 1st
enough resources then for P2
Unsafe vs. Safe

- State C could follow State B
- P2 requests R2, just assign it
- Bad Philosophers: each gets one R
- **cycle** (assign/claim edges)
- but NOT deadlock
- no SS, hence no guarantee of future
- **Avoidance**: UNDO, do NOT assign
- always stay safe, no deadlock

OR: C’ follows B
- P1 requests R2, assign
- Good Philosophers
- no cycle
- SS: P1, P2 obviously safe, go ahead
• if we do NOT UNDO C:
  • i.e. not using Avoidance
  • State D could follow C
  • P1 requests R2, block
  • guess what's coming?

OR: D’ could follow C
P2 finished with R2
State E could follow D
- P2 requests R1, block
- Bad Philosophers
- deadlock
- cycle (assign/request)
- deadly embrace
- Avoidance would have stopped C

State F’ could follow D’
- P1 requests R2, assign
- P1 can finish its work
- no problem
- is E or F’ the future?
- Avoidance: stay safe, stop C
Avoidance: Conclusions

- cycle (assign/claim) may lead to cycle (assign/blocked request)
- but Avoidance does not know where the system is headed
- Avoidance prevents first cycle, to guarantee no second cycle
- second cycle is deadlock
- preventing first cycle guarantees a SS (need only one SS for proof)
- Avoidance stops potential problems early
- lower resource utilization, but guaranteed no deadlock
Banker’s Algorithm

- multiple instances of resource types ⇒ cannot use resource-allocation graph
- banks do not allocate cash unless they can satisfy customer needs
- when a new process enters the system
declare in advance maximum need for each resource type
cannot exceed the total resources of that type
- later, processes make actual request for some resources
- if the allocation leaves system in safe state
  grant the resources
- otherwise
  suspend process until other processes release enough resources
**Example of Avoidance**

- must specify Max Need in advance, same as **claim**
- Allocation the same as **assigned**
- Need is the difference between the above, how much room to grow
- assume 12 tape drives: Multiple Instance (any one will do)

<table>
<thead>
<tr>
<th>Max Need</th>
<th>Current Alloc</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>
[V] Safe Sequence

- sequence \( <P_1, P_0, P_2> \) is a **safe sequence**
- \( P_1 \) only needs at most 2 more, and 3 are available. Alloc 2 more:

\[
\begin{array}{ccc}
\text{Max Need} & \text{Current Alloc} & \text{Need} & \text{Available} \\
4 & 4 & 0 & 1 \\
\end{array}
\]

- assume \( P_1 \) now finishes and gives up all 4 resources:

\[
\begin{array}{ccc}
\text{Max Need} & \text{Current Alloc} & \text{Need} & \text{Available} \\
4 & 0 & 0 & 5 \\
\end{array}
\]

- we just moved Available to Alloc, back to Available
- why not just move the original Current Allocation (2) to Available?
- end up in the same state
- key: is there a process with \( \text{Need} \leq \text{Available} \)?
- if yes, then dump their Alloc into Available, repeat.
- if no, then no safe sequence (SS)
## Proof of Safe Sequence

<table>
<thead>
<tr>
<th></th>
<th>Max Need</th>
<th>−</th>
<th>Current</th>
<th>Alloc</th>
<th>=</th>
<th>Need</th>
<th>≤</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>4</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
No Safe Sequence

- note Available (12) is back to the original system total
- We are not really running these processes in this order!
- we are just checking if a safe sequence exists
- now assume $P_2$ requests one more tape drive, try allocation:

<table>
<thead>
<tr>
<th>Max Need</th>
<th>Current Alloc</th>
<th>= Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

- $P_1$ has an Alloc (2) $\leq$ Available (2)
- returning this to Available, makes 4, but no other Need $\leq$ 4
- no safe sequence
- Do not make this allocation, remove it!
- graph shows state for one particular process $P_i$ from 0..$N - 1$
- $Max_{ij}$: $P_i$ claims it might need of each resource $R_j$ from 0..$M - 1$
- $Alloc_{ij}$: $P_i$ already allocated this amount of resource $R_j$
- $Need_{ij}$: $Max_{ij} - Alloc_{ij}$ how much more can be requested
- since one particular $i$, looking at vectors sliced from these matrices
Banker: Data Structures

#define MAXN 10  /* maximum number of processes */
#define MAXM 10  /* maximum number of resource types */
int Available[MAXM];  /* Available[j] = current # of unused resource j */
int Max[MAXN][MAXM];  /* Max[i][j] = max demand of i for resource j */
int Allocation[MAXN][MAXM];  /* Allocation[i][j] = i's current allocation of j*/
int Need[MAXN][MAXM];  /* Need[i][j] = i's potential for more j */
  /* Need[i][j] = Max[i][j] - Allocation[i][j] */

Notation:

$ X \leq Y \iff X[i] \leq Y[i] \text{ for all } i$

$(0,3,2,1)$ is less than $(1,7,3,2)$

$(1,7,3,2)$ is NOT less than $(0,8,2,1)$

Each row of $Allocation$ and $Need$ are vectors: $Allocation_i$ and $Need_i$
Safety Algorithm

- consider some sequence of processes
- if the first process has $Need$ less than $Available$
  - it can run until done
  - then release all of its allocated resources
    - allocation is increased for next process
- if the second process has $Need$ less than $Available$
- ...
- then all of the processes will be able to run eventually
- $\Rightarrow$ system is in a safe state
STEP 1: initialize
   Work := Available;
   for i = 1,2,...,n
       Finish[i] = false

STEP 2: find i such that both
   a. Finish[i] is false
   b. Need_i <= Work
   if no such i, goto STEP 4

STEP 3:
   Work := Work + Allocation_i
   Finish[i] = true
   goto STEP 2

STEP 4:
   if Finish[i] = true for all i, system is in safe state
**Full Banker’s Algorithm**

**STEP 0:** $P_i$ makes Request$_i$ for resources, say $(1,0,2)$

**STEP 1:** if Request$_i$ $<=$ Need$_i$

    goto STEP 2

    else ERROR

**STEP 2:** if Request$_i$ $<=$ Available

    goto STEP 3

    else suspend $P_i$

**STEP 3:** pretend to allocate requested resources

    Available := Available - Request$_i$

    Allocation$_i$ := Allocation$_i$ + Request$_i$;

    Need$_i$ := Need$_i$ - Request$_i$

**STEP 4:** if pretend state is SAFE

    then do a real allocation and $P_i$ proceeds

    else

    restore the original state and suspend $P_i$
Initially:
Available
  A B C
  10 5 7
Later Snapshot:

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

- this state is safe (Banker always keeps it safe)
- \(< P_1, P_3, P_4, P_2, P_0 >\) satisfies criteria
- proof follows
State Example

- these examples are from Silberschatz
- there could be many safe sequences
- using FOR loops, algorithm would find "lower" indices first
- say two processes both have Need \leq Available
- does not matter which one appears first in safe sequence
- both just increase the "working" available
### Safety Example

<table>
<thead>
<tr>
<th>Max</th>
<th>Allocation</th>
<th>Need &lt;= Work</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P1</td>
<td>3 2 2</td>
<td>2 0 0</td>
<td>1 2 2 &lt;= 3 3 2 &lt;= 3 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+2 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1 &lt;= 5 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+2 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1 &lt;= 7 4 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 0 2</td>
</tr>
<tr>
<td>P2</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0 &lt;= 7 4 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+3 0 2</td>
</tr>
<tr>
<td>P0</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3 &lt;= 10 4 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 5 7&lt;&lt;&lt;&lt; initial system</td>
</tr>
</tbody>
</table>
Full Banker Example

- Say $P_1$ requests (1,0,2)
- Compare to $\text{Need}_1$: (1,0,2) $\leq$ (1,2,2)
- Compare to $\text{Available}$: (1,0,2) $\leq$ (3 3 2)
- Pretend to allocate resources:
  
<table>
<thead>
<tr>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 2 2</td>
<td>3 0 2&lt;&lt;&lt;</td>
<td>0 2 0&lt;&lt;&lt;</td>
</tr>
<tr>
<td>P2</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>
- Is this safe? Yes: $< P_1, P_3, P_4, P_0, P_2 >$
- proof follows
<table>
<thead>
<tr>
<th>P1</th>
<th>3 2 2</th>
<th>3 0 2</th>
<th>0 2 0</th>
<th>2 3 0</th>
<th>2 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td>5 3 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td>7 4 3</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td>7 4 5</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td>7 5 5</td>
<td></td>
</tr>
</tbody>
</table>

10 5 7 <<< initial system
**More Examples**

- Can $P_4$ get $(3,3,0)$? No, $(3,3,0) > (2,3,0)$ Available
- Can $P_0$ get $(0,2,0)$? $(0,2,0) < (2,3,0)$ Available
- Pretend: Available goes to $(2,1,0)$
- But ALL Needs are greater than Available $\Rightarrow$ NOT SAFE
[VI] Memory Management

- CPU runs program instructions only when program is in memory
- programs do I/O sometimes ⇒ CPU wasted
- solution: multiprogramming (multitasking)
  - multiple programs (processes) share the memory
  - one program, at at time, gets CPU
  - simultaneous resource possession (CPU and memory)
  - better performance (response time, throughput)
- will study: A) Main Memory and B) Virtual Memory
**VI.A** Long/Short Term Schedulers

- **long-term**: job scheduler (memory management)
  - *which jobs allocated memory and allowed into the system*
- **short-term**: CPU scheduler (process management)
  - *which job allocated the CPU*
[VI.A] Memory: Storage Hierarchy

- CPU
  - registers
  - control store
    - local cache
  - global cache
    - main memory
      - I/O cache
        - fixed-head disk
        - removable disk
        - magnetic tapes
<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Capacity</th>
<th>Access Time</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Store</td>
<td>1K-32K words</td>
<td>3-100 ns</td>
<td>SRAM, ROM</td>
</tr>
<tr>
<td>Registers</td>
<td>8-256 words</td>
<td>3-10 ns</td>
<td>SRAM</td>
</tr>
<tr>
<td>CPU Cache</td>
<td>8 KB - 1 MB</td>
<td>3-100 ns</td>
<td>SRAM, DRAM</td>
</tr>
<tr>
<td>Main Memory</td>
<td>128 KB - 4 GB</td>
<td>20-200 ns</td>
<td>DRAM</td>
</tr>
<tr>
<td>I/O Cache</td>
<td>32 KB - 1 MB</td>
<td>20-200 ns</td>
<td>DRAM</td>
</tr>
<tr>
<td>Disk Drive</td>
<td>1 MB - 100 GB</td>
<td>10-65 ms</td>
<td>Magnetic disk</td>
</tr>
<tr>
<td>Tape Drive</td>
<td>20 MG or more</td>
<td>seconds</td>
<td>Magnetic tape</td>
</tr>
</tbody>
</table>
[VI.A] Memory: Overlays

- Program does pass 1; then pass 2
- Programmer controls memory
- Used when physical memory size was limited
**[VI.A] Memory: Dynamic Relocation**

- **logical address**: as seen by CPU
- **physical address**: as seen by memory unit
- to relocate program/data: move and change register
  - save both registers during a context switch
**VI.A** Memory: Swapping

- **swap-out**: executing job (round-robin)
- **swap-in**: old job (maybe dynamic relocation)
- **roll-out, roll-in**: low priority for high priority
- 2 jobs \( \times (100K \text{ size} \times 1000K/\text{sec} + 8\text{ms latency}) = 216\text{ms} \)
- quantum: much larger than 216ms
[VI.A] Memory: Contiguous Allocation

- program and data space are in sequential memory addresses
- single-partition: OS and one program
  - relocation register and limit register
  - protect OS and user program from each other
- multiple-partitions: many programs with their own partition
  - required for multi-programming
  - fixed-partition scheme (IBM OS/360 MFT)
  - variable-partition scheme (IBM OS/360 MVT)
## [VI.A] Memory: Scheduling Example

<table>
<thead>
<tr>
<th>process</th>
<th>job queue</th>
<th>memory</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>600K</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1000K</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>$P_3$</td>
<td>300K</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>$P_4$</td>
<td>700K</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>$P_5$</td>
<td>500K</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td>58</td>
</tr>
</tbody>
</table>

**Scheduling Discipline:**
- **Job:** FCFS
- **CPU:** Round-Robin (Quantum=1)
- **Memory:** 2560 KB

**Dynamic Storage Allocation Strategy:** Best fit
### Memory: Scheduling Details

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_5$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>39</th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
<th>46</th>
<th>47</th>
<th>48</th>
<th>49</th>
<th>50</th>
<th>51</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_5$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>$P_3$</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>18</td>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>53</th>
<th>54</th>
<th>55</th>
<th>56</th>
<th>57</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_5$</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$P_3$</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
[VI.A] Memory: Partitions

<table>
<thead>
<tr>
<th>Time</th>
<th>Process</th>
<th>Memory Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=14</td>
<td>OS</td>
<td>0-400K</td>
</tr>
<tr>
<td>400K</td>
<td>P1</td>
<td>400K-1000K</td>
</tr>
<tr>
<td>t=14</td>
<td></td>
<td>400K</td>
</tr>
<tr>
<td>1000K</td>
<td>P2</td>
<td>1000K-2000K</td>
</tr>
<tr>
<td>P2 terminates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000K</td>
<td>P3</td>
<td>2000K-2300K</td>
</tr>
<tr>
<td>P4 allocate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2300K</td>
<td>P3</td>
<td>2300K-2560K</td>
</tr>
<tr>
<td>2560K</td>
<td>P3</td>
<td>2560K-</td>
</tr>
<tr>
<td>t=28</td>
<td>OS</td>
<td>0-400K</td>
</tr>
<tr>
<td>400K</td>
<td>P1 terminates</td>
<td>400K-1000K</td>
</tr>
<tr>
<td>P4 allocate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000K</td>
<td>P4</td>
<td>1000K-1700K</td>
</tr>
<tr>
<td>1700K</td>
<td>P5 allocate</td>
<td>1700K-</td>
</tr>
<tr>
<td>P5</td>
<td></td>
<td>1700K-900K</td>
</tr>
<tr>
<td>900K</td>
<td>P5</td>
<td>900K-</td>
</tr>
<tr>
<td>1000K</td>
<td>P5</td>
<td>1000K-</td>
</tr>
<tr>
<td>1700K</td>
<td>P5</td>
<td>1700K-</td>
</tr>
<tr>
<td>2000K</td>
<td>P3</td>
<td>2000K-</td>
</tr>
<tr>
<td>2300K</td>
<td>P3</td>
<td>2300K-</td>
</tr>
<tr>
<td>2560K</td>
<td>P3</td>
<td>2560K-</td>
</tr>
</tbody>
</table>
### Memory: Search Strategies

- list of memory holes: which one to use?
- first-fit: allocate the first hole which is big enough
  - produces the smallest left-over hole
- best-fit: allocate the smallest hole which is big enough
  - produces the smallest left-over hole
- worst-fit: allocate the largest hole
  - produces the largest left-over hole

<table>
<thead>
<tr>
<th>strategy</th>
<th>search time</th>
<th>memory utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>first-fit</td>
<td>fast</td>
<td>good</td>
</tr>
<tr>
<td>best-fit</td>
<td>slow</td>
<td>good</td>
</tr>
<tr>
<td>worst-fit</td>
<td>slow</td>
<td>bad</td>
</tr>
</tbody>
</table>
[VI.A] Memory: External Fragmentation [111]

- request for memory cannot be satisfied
  - even though total free memory is sufficient
  - but not contiguous (broken into small holes)
- first-fit or best-fit may be better
- 50-percent rule:
  - given $N$ allocated blocks $\Rightarrow 0.5N$ blocks unusable ($1/3$ wasted)
- solution: compaction or paging
**VI.A** Memory: Internal Fragmentation

- allocated memory slightly larger than requested memory
- overhead to keep track of small hole may be larger than hole itself

Diagram:

```
OS
P7
P43
```

next request: 18,462 bytes

hole of 18,464 bytes
- move 600K, or 400K, or 200K
- dynamic relocation and swapping
[VI.A] Memory: Paging

- goal: eliminate external fragmentation
- (allow noncontiguous processes)
- each process is a set of fixed-size pages
- pages are stored in same-size physical memory “frames”
- page table “connects” logical pages with physical frames
- may still have internal fragmentation
[VI.A] Memory: Paging Hardware

- **logical address**: $p$ (page) and $d$ (displacement/offset) in page
- **physical address**: $f$ (frame) and $d$ (displacement/offset) in frame
- **PTBR**: Page Table Base Register
- **PTLR**: Page Table Length Register
- byte 'g': logical address = 1 10 = 6
- byte 'g': physical address = 101 10 = 22
[VI.A] Memory: Logical Address

- page size: $2^n$
- logical address space: $2^m$
- page number: high-order $m - n$ bits
- page offset: low-order $n$ bits

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$d$</td>
</tr>
<tr>
<td>$m - n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
VI.A Memory: Logical Addresses

- \( n = 10 \) and \( m = 32 \)

```c
typedef union {
    struct {
        unsigned int page : 22;
        unsigned int offset : 10;
    } logical;
    int addr;
} laddr;
```

- page 0: 0..1023
- page 1: 1024..2047
- \( 2^{m-n} = 4,194,304 \)
- \( l = 1026; \) /* page=1, offset=2 */
VI.A] Memory: Segmentation

- supports user view of memory
- logical address space is a collection of segments (name and length)
- address = [segment name or number] [offset]
- closely related to partitions (but several per program)
V.A] Memory: Paging vs. Segmentation

- paging
  - no external fragmentation
  - large tables
- segmentation
  - external fragmentation
  - small tables
  - easy protection at segment level
  - easy sharing at segment level
- solution: paged segmentation
[VI.B] Virtual Memory: Introduction

- up to now: all of a process in main memory (somewhere)
  - partitions, pages, segments
- now: virtual memory
  - allow execution of processes which may be partially in memory
- benefits:
  - programs can be large and memory can be small
  - increased multiprogramming \(\Rightarrow\) better performance
  - less I/O for loading/swapping programs
- why programs don’t need to be entirely in memory:
  - code for unusual error conditions
  - more memory allocated than is needed
  - some features of program rarely used
- VM is the separation of user logical memory from physical memory
[VI.B] VM: Page Table

- **logical address space**
- **physical address space**
- **paging device**

- logical address vs. physical address
- demand paging: only “necessary” pages are brought into memory
- page table: points to frames
- frame table: points to pages
- (this implementation only works with one process)
- bit: is page loaded into a frame?
[VI.B] VM: Page Fault

1. reference
2. trap
3. find free frame
4. read in page
5. reset page table
6. restart

3a. find page on disk

OS

Load M

Page Table

Physical Memory
[VI.B] VM: Page Fault Details

- trap to OS - save user registers and process state
- determine that interrupt was a page fault
- check page reference and location on disk
- issue read from the disk to a free frame (old page not DIRTY)
  - wait in queue for device
  - wait for seek and/or latency
  - begin transfer
- while waiting, allocate CPU to other user
- interrupt from the disk (I/O completed)
- save registers and process state of other user
- determine that interrupt was from disk
- correct page table (desired page is now in memory)
- wait for CPU to be allocated to this process again
- restore user registers, process state, new page table
- RESUME execution
[VI.B] VM: FIFO (replace the oldest page)  [126]

mem> alg 0
mem> mode 1
mem> init 8 3
mem> r 7 r 0 r 1 r 2 r 0 r 3 r 0 r 4 r 2 r 3 r 0 r 3 r 2 r 1 r 2 r 0 r 1 r 7 r 0 r 1

page: 7  faults: 1  frames: 7 -1 -1  time: 1
0    2    7  0 -1  1  2
1    3    7  0 1  1  2  3
2    4    2  0 1  4  2  3
0    4    2  0 1  4  2  3
3    5    2  3 1  4  6  3
0    6    2  3 0  4  6  7
4    7    4  3 0  8  6  7
2    8    4  2 0  8  9  7
3    9    4  2 3  8  9 10
0    10   0  2 3  11 9 10
3    10   0  2 3  11 9 10
2    10   0  2 3  11 9 10
1    11   0  1 3  11 14 10
2    12   0  1 2  11 14 15
0    12   0  1 2  11 14 15
1    12   0  1 2  11 14 15
7    13   7  1 2  18 14 15
0    14   7  0 2  18 19 15
1    15   7  0 1  18 19 20
VI.B VM: Optimal Algorithm

- “replace the page that will not be used for the longest time”
- lowest page-fault rate of all algorithms
- requires advanced knowledge of page reference string
- useful for comparison studies
[VI.B] VM: OPT

mem> nogo
mem> r 7 r 0 r 1 r 2 r 0 r 3 r 0 r 4 r 2 r 3 r 0 r 3 r 2 r 1 r 2 r 0 r 1 r 7 r 0 r 1
mem> go

    page:  7  faults:  1  frames:  7  -1  -1
      0     2    7  0  -1
      1     3    7  0  1
      2     4    2  0  1
      0     4    2  0  1
      3     5    2  0  3
      0     5    2  0  3
      4     6    2  4  3
      2     6    2  4  3
      3     6    2  4  3
      0     7    2  0  3
      3     7    2  0  3
      2     7    2  0  3
      1     8    2  0  1
      2     8    2  0  1
      0     8    2  0  1
      1     8    2  0  1
      7     9    7  0  1
      0     9    7  0  1
      1     9    7  0  1
[VI.B] VM: Least Recently Used (LRU) [129]

- FIFO: when a page was brought into memory in the past
- OPT: when a page is used in the future
- LRU: when a page was used in the past
  - “replace the page that has not been used for the longest time”
  - requires a logical clock time for each page (LRU_TIME)
  - or a stack of recent page references (LRU_STACK)
  - or a list of all references (LRU_REF)
    - same as OPT but on reverse of page reference string
    - optimal algorithm looking backward in time
**[VI.B] VM: LRU_TIME**

- better than FIFO (15) but worst than OPT (9)

<table>
<thead>
<tr>
<th>page</th>
<th>7</th>
<th>faults: 1</th>
<th>frames: 7 -1 -1</th>
<th>time: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td>7 0 -1</td>
<td>1 2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td>7 0 1</td>
<td>1 2 3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td>2 0 1</td>
<td>4 2 3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td>2 0 1</td>
<td>4 5 3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td>2 0 3</td>
<td>4 5 6</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td>2 0 3</td>
<td>4 7 6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
<td>4 0 3</td>
<td>8 7 6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
<td>4 0 2</td>
<td>8 7 9</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td>4 3 2</td>
<td>8 10 9</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td></td>
<td>0 3 2</td>
<td>11 10 9</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td>0 3 2</td>
<td>11 12 9</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
<td>0 3 2</td>
<td>11 12 13</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td>1 3 2</td>
<td>14 12 13</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
<td>1 3 2</td>
<td>14 12 15</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td></td>
<td>1 0 2</td>
<td>14 16 15</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td></td>
<td>1 0 2</td>
<td>17 16 15</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td></td>
<td>1 0 7</td>
<td>17 16 18</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td></td>
<td>1 0 7</td>
<td>17 19 18</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td></td>
<td>1 0 7</td>
<td>20 19 18</td>
</tr>
</tbody>
</table>
[VI.B] VM: LRU_STACK

- keep a stack of every page which owns a frame
- on each reference
  - find the page within the stack and remove it
  - push the page onto the top of the stack
- on page fault
  - if FREE frame, take it
  - otherwise, LRU page is at the bottom of the stack
  - return its frame
## [VI.B] VM: LRU_STACK

<table>
<thead>
<tr>
<th>page</th>
<th>faults</th>
<th>frames</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
• new page reference : 4
• current frame pages: 2 0 3
• current ref string : 7 0 1 2 0 3 0 4
• reverse ref string : 4 0 3 0 2 1 0 7
  ↑
  ↑
  ↑
• apply OPT algorithm: 2 0 3 → 4 0 3
[VI.B] VM: LRU Approximations

- hardware usually does not support LRU
- but does support REF bit
  - interrupt every 100 msec
  - move REF bit to 8-bit shift register (and clear)
  - 00000000 \( \Rightarrow \) no refs in last 8 periods
  - 11111111 \( \Rightarrow \) at least one ref in each period
  - 01111111 \( \Rightarrow \) no ref in the recent period
  - smallest integer \( \Rightarrow \approx \) LRU
  - “additional-reference-bits algorithm”
- ONE bit of history
  - just use the REF bit itself
  - “second-chance” page-replacement algorithm
[VI.B] Second-Chance (CLOCK) Algorithm [135]

- if REF then clear bit (NOREF) and move on
- next victim is a NOREF
- if all pages have been referenced then CLOCK = FIFO
- frame table is circular queue of pages
better than FIFO (15) but worse than LRU (12)

<table>
<thead>
<tr>
<th>page</th>
<th>faults</th>
<th>frames</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>7 0 -1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7 0 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>2 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2 0 3</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>2 0 3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4 0 3</td>
<td>1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>4 2 3</td>
<td>1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4 2 3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>4 2 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>3 2 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>3 2 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>3 1 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>3 1 2</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>0 1 2</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>0 1 2</td>
<td>1 1 1</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>0 7 2</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0 7 2</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>0 7 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>
[V.B] VM: ENHANCED Second Chance

- if victim is dirty then it costs time to write it out
- better to choose a non-dirty victim (Macintosh VM)

  class1: (ref=0, dirty=0) => good page to replace
  class2: (0,1) => not as good because old page needs to be written
  class3: (1,0) => not good because its recently referenced
  class4: (1,1) => definitely not good because it also has to be written

PASS: do
  a. if empty frame, take it
  b. if class1, take it
  c. if class2, then record first instance
  d. clear ref bit if class 2 has not been found yet
until complete pass
if class2 was found, take first instance
invariant1: there are no free frames
invariant2: there are only class1 and class2 because all bits were cleared.

if the first PASS does not succeed, try one more PASS
if no writing, ENHANCED = CLOCK

in the case below, WRITES alter the sequence of events

| rpage: 7 faults: 1 frames: 7 -1 -1 ref,dirty: 1,0 |
|-------|-------|-------|-------|
| 0     | 2     | 7     | 0     | -1   | 1,0 | 1,0 |
| 1     | 3     | 7     | 0     | 1    | 1,0 | 1,0 | 1,1 |
| 2     | 4     | 2     | 0     | 1    | 1,0 | 0,0 | 0,1 |
| 0     | 4     | 2     | 0     | 1    | 1,0 | 0,0 | 0,1 |
| 3     | 5     | 2     | 0     | 3    | 1,0 | 0,0 | 1,0 |
| 4     | 6     | 4     | 0     | 3    | 1,0 | 0,1 | 0,0 |
| 2     | 7     | 4     | 0     | 2    | 1,0 | 0,1 | 1,0 |
| 3     | 8     | 4     | 3     | 2    | 0,0 | 1,0 | 1,0 |
| 0     | 9     | 0     | 3     | 2    | 1,0 | 1,0 | 0,0 |
| 3     | 9     | 0     | 3     | 2    | 1,0 | 1,0 | 0,0 |
| 2     | 9     | 0     | 3     | 2    | 1,0 | 1,0 | 1,0 |
| 1     | 10    | 0     | 1     | 2    | 0,0 | 1,0 | 0,0 |
| 2     | 10    | 0     | 1     | 2    | 0,0 | 1,0 | 1,0 |
| 0     | 10    | 0     | 1     | 2    | 1,0 | 1,0 | 1,0 |
| 1     | 10    | 0     | 1     | 2    | 1,0 | 1,0 | 1,0 |
| 7     | 11    | 0     | 1     | 7    | 0,0 | 0,0 | 1,0 |
| 0     | 11    | 0     | 1     | 7    | 1,0 | 0,0 | 1,0 |
| 1     | 11    | 0     | 1     | 7    | 1,0 | 1,0 | 1,0 |
[VI.B] VM: Counting Algorithms

- maintain a count of the number of references to a page
- Least Frequently Used (LFU): “replace page with smallest count”
  - active pages will have large counts
  - but initialization pages will have large counts
- Most Frequently Used (MFU): “replace page with largest count”
  - recent pages are active pages but will have small counts
- neither are common
- do not approximate OPT well
### [VI.B] VM: LFU

<table>
<thead>
<tr>
<th>page</th>
<th>faults</th>
<th>frames</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>page</td>
<td>faults</td>
<td>frames</td>
<td>count</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>7 0 -1</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7 0 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2 0 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>2 0 1</td>
<td>1 2 1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2 3 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>2 3 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>4 3 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>4 2 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>4 2 3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0 2 3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0 2 3</td>
<td>1 1 2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0 2 3</td>
<td>1 2 2</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0 1 3</td>
<td>1 1 2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0 1 2</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>0 1 2</td>
<td>2 1 1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>0 1 2</td>
<td>2 2 1</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>7 1 2</td>
<td>1 2 1</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>7 0 2</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>7 0 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>
[VI.B] Replacement Algorithm Performance [143]

- more frames ⇒ less faults
- if [frames allocated] = 1?
- if [frames allocated] = [logical pages]?

mem> mode 2
mem> init 8 8
- more frames ⇒ more faults
- FIFO may replace pages that are just about to be used again
### VM: Belady’s Anomaly

<table>
<thead>
<tr>
<th>page</th>
<th>faults</th>
<th>frames</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 -1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1 2 -1</td>
<td>1 2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1 2 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4 2 3</td>
<td>4 2 3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>4 1 3</td>
<td>4 5 3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4 1 2</td>
<td>4 5 6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>5 1 2</td>
<td>7 5 6</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5 1 2</td>
<td>7 5 6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5 1 2</td>
<td>7 5 6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5 3 2</td>
<td>7 10 6</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>5 3 4</td>
<td>7 10 11</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>5 3 4</td>
<td>7 10 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page</th>
<th>faults</th>
<th>frames</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 -1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1 2 -1</td>
<td>1 2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1 2 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5 2 3 4</td>
<td>7 2 3 4</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>5 1 3 4</td>
<td>7 8 3 4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5 1 2 4</td>
<td>7 8 9 4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5 1 2 3</td>
<td>7 8 9 10</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>4 1 2 3</td>
<td>11 8 9 10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>4 5 2 3</td>
<td>11 12 9 10</td>
</tr>
</tbody>
</table>
[VI.B] VM: Allocation of Frames

- up to now: just one process in memory and all frames are available
- $m$ frames available for $n$ processes
- equal allocation: $m/n$ frames per process
  - but small processes may get too many frames
- proportional allocation based on size of process: $s_i/S 	imes m$
  - $S = \sum s_i$
  - may want to increase allocation for high-priority processes
- what happens if a fault occurs and no free frames?
  - local replacement: reuse a frame from the faulting process
    - can not make use of under-utilized frames
  - global replacement: take a frame from any process
    - high-priority takes from low-priority ("stealing")
    - "fate" of a process depends on the behavior of others
    - tends to increase system throughput
    - but may cause thrashing
**VI.B** VM: Thrashing

- **scenario 1:**
  - process has a small number of frames (allocation or stealing)
  - process has a large number of active pages
  - process spends more time paging than executing

- **scenario 2:**
  - OS monitors CPU utilization
  - if low utilization then increase degree of multiprogramming
  - new process takes frames from other processes
    - they start thrashing
  - utilization decreases and OS adds more processes
  - more thrashing
[VI.B] VM: Thrashing

CPU utilization

degree of multiprogramming

thrashing
[VI.B] VM: Thrashing Solutions

- use a local replacement algorithm
  - thrashing process cannot steal frames
  - but queue time (paging device) will increase for ALL processes
- provide a process as many frames as it “needs”
  - may suspend other processes (and free up their frames)
  - “need” based on *locality model*
    - set of pages that are actively used together
    - subroutines or data structures
    - if full set in memory then no more faults (until new locality)
[VI.B] VM: Working-Set Model

- approximation to the program’s locality
- let $\Delta$ be the working-set window
- keep list of all pages used during the last $\Delta$ page references

\[ \text{window} \quad \text{window} \]

\[ \text{WS} = 1,2,5,6,7 \quad \text{WS} = 3,4 \]

- small $\Delta \Rightarrow$ does not encompass locality
- large $\Delta \Rightarrow$ given too many frames
- let $WSS_i$ be the size of the working set for process $i$
- demand $D = \sum WSS_i$
- if not enough frames, then suspend processes (and free frames)
- prevents thrashing and keeps multiprogramming high as possible
- optimizes CPU utilization

- PFF Strategy
- direct approach to solve thrashing
- may need to suspend other processes to get more frames

![Diagram](image)

- number of page faults
- upper bound
- lower bound
- increase number of frames
- decrease number of frames

number of page frames allocated
Disk Scheduling

- A physical device like a disk is slow compared to CPU.
- To access a block (say 512 bytes):
  - Disk-arm moves to appropriate radius: seek time (11ms)
  - Spinning disk (platter) rotates to position: latency time (13ms)
  - Read-write head copies data: transfer time (53 micro)
- During one read/write, many new block requests may arrive.
- So the OS knows in advance a batch of block requests.
- How can a disk scheduling algorithm help improve the seek time?
• First-Come First-Served: process blocks in the order of arrival
• while processing block 53, nine more requests arrive
• process the nine requests: 98, 183, 37, 122, 14, 124, 65, 67, 190
• long disk-arm movements back-and-forth, poor seek time
• total distance $d = 763$ (53..98..183..37..)
• not a good scheduling algorithm
• Shortest-Seek Time First: greedy, not necessarily optimal
• move arm the shortest possible distance to nearest block request
• 65, 67, 37, 14, 98, 122, 124, 183, 190
• $d = 243 \ (53..65..67..37..)$
• if 12 arrives before finishing 14, arm would move backwards
• blocks farther away not serviced due to new/close blocks
• this could repeat, causing starvation/unfairness
- SCAN is the first of 4 elevator algorithms
- like an elevator, move up/down (actually inside/outside)
- process people (block requests) as it goes 0..199..0..199..
- assume arm is moving upwards as it encounters 53
- keep on going all the way to end of disk 199, and back to 0
- 65, 67, 98, 122, 124, 183, 190, (199), 37, 14 (and then 0, repeat)
- \( d = 146 \times (53..199) + 185 \times (199..14) = 331 \)
[VII] Disk: SCAN

- arm does not have jerky motion of FCFS or SSTF
- may not be optimal either
- assume 75 arrives while processing 124
- it will be processed on the way down
- but before 37, 14 which are older
- unfair, but will not starve because arm keeps moving same direction
- how to solve this unfairness: do not process blocks on way down
• C-SCAN: Circular-SCAN, do no process blocks on way down
• in this case, 14, 37 will be processed on the way back up
• 65, 67, 98, 122, 124, 183, 190, (199), (0), 14, 37
• d = 146 (53..199) + 199 (199..0) + 37 (0..37) = 382
what if 75 arrives while processing 124?
- it will be processed last, after 37, so fair
- what if 75 arrives while processing 67?
- it will be processed right afterwards because arm is moving up
- it is processed before older 98, so still some unfairness
- SCAN algorithms move to edges even though no blocks to process
- these algorithms are meant as examples
- get rid of extra movement: LOOK
just like SCAN but only go as far as necessary in a direction

- 65, 67, 98, 122, 124, 183, 190 (turn-around), 37, 14 (turn-around)
- $d = 137 (53..190) + 176 (190..14) = 313$
- with respect to 75, still has same unfairness as SCAN
- if 195 arrives just after the 190 turn-around, it will wait long
- in some cases, good to pick-up close block by changing directions
- so elevator algorithms are smooth, but not optimal either
**C-LOOK**: Circular-LOOK, same as C-SCAN but don’t go to edges

- $65, 67, 98, 122, 124, 183, 190$ (turn-around), $14$ (reset), $37$
- $d = 137 (53..190) + 176 (190..14) + 23 (14..37) = 336$
## Index to Slides: Operating Systems

<table>
<thead>
<tr>
<th>I. Process Scheduling</th>
<th>V. Deadlock</th>
<th>B. Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCB</td>
<td>Prevention</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Detection</td>
<td>FIFO</td>
</tr>
<tr>
<td>FCFS</td>
<td>Recovery</td>
<td>OPT</td>
</tr>
<tr>
<td>RR</td>
<td>Avoidance</td>
<td>LRU</td>
</tr>
<tr>
<td>SJF</td>
<td>Allocation Graph</td>
<td>CLOCK</td>
</tr>
<tr>
<td>Prio</td>
<td>Banker's Algorithm</td>
<td>LFU/MFU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Belady's Anomaly</td>
</tr>
<tr>
<td>II. Semaphores</td>
<td>VI. Memory Management</td>
<td></td>
</tr>
<tr>
<td>Critical Section</td>
<td>A. Main Memory</td>
<td></td>
</tr>
<tr>
<td>Semaphores</td>
<td>CPU/Job Schedulers</td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td>Address Binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thrashing</td>
</tr>
<tr>
<td>III. Classical Problems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philosophers</td>
<td>Dynamic Relocation</td>
<td></td>
</tr>
<tr>
<td>Producer/Consumer</td>
<td>Partitions</td>
<td></td>
</tr>
<tr>
<td>Readers/Writer</td>
<td>Fragmentation</td>
<td></td>
</tr>
<tr>
<td>IV. Progress/Fairness</td>
<td>Paging</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logical Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Segmentation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Working Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VII. Disk Scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FCFS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSTF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C-SCAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOOK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C-LOOK</td>
</tr>
</tbody>
</table>