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

CS 6560

Operating Systems Design (Theory)

Instructor: Ted Billard

Operating Systems: Outline [2]

[I] Process Scheduling [3]

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

[I] Process: Definitions [4]

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** • executes processes, performing the instructions

[I] Process: Definitions [5]

running
- one process is **running**, or executing, on the CPU

concurrent
- time-share CPU - appear to run simultaneously

cooperating
- interact and affect each other
- share data or resources
- **synchronous**
- **asynchronous**: not cooperating (no interaction)

state
- current activity or condition (e.g. running)

suspended
- waiting for memory - then joins the **ready queue**

[I] Process: Definitions [6]

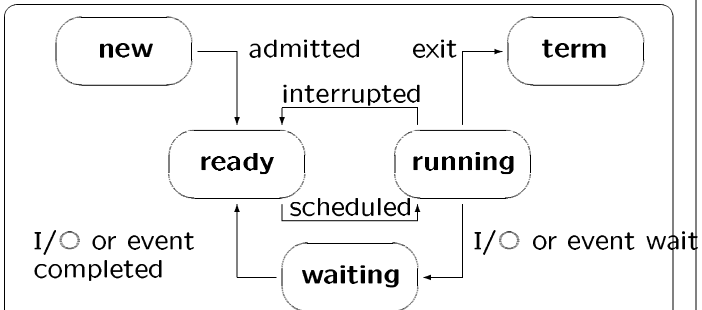
ready
- waiting for access to the CPU
- not waiting for I/O or some other event
- in the **ready queue** - list of process IDs (PIDs)
- ready → running: **scheduled** for CPU
- running → ready: **interrupted** by scheduler
- running → **waiting**

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**

[I] Process: State Changes [7]

Figure 4.1



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

[I] Process Control Block (PCB) [8]

Figure 4.2

pointer
process state
process number
program counter
registers
memory limits
list of open files
⋮

- 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 PCB_1
  - put PID_1 back in ready queue (or other queue)
  - remove PID_2 from ready queue, load registers based on PCB_2
  - start executing PID_2 at **pc** register

[I] Program Counter (pc) [9]

```
     |               |               |              \
      \             /           pc-> |              /
       |            \                |             /
       |             |                \       pc-> \
      /              |               /              \
pc-> |              /               /               /
    /               \              /               /
   |                 \            /                \
   |            pc-> \            \                 \
    \                 /            \               /
```

- each **process** (or **thread**) has a **pc**
- increments **sequentially** but branches for loops and conditionals

[I] Context Switch by Dispatcher [10]

```
executing |                                           .
          v --------------> dispatcher                . idle
          .                 SAVE: PCB 1               .
          .                 LOAD: PCB 2 ------------> |
     idle .                                 executing |
          .                                           |
          .                 dispatcher  <------------ v
          .                 SAVE: PCB 2               .
          | <-------------- LOAD: PCB 1               . idle
executing v                                           .
```

- CPU is switched to another process
- state (**PCB**) of the old process is saved
- saved state of the new process is loaded

[I] Interrupts [11]

- *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

[I] Interrupts [12]

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

[I] CPU Scheduling [13]

summary	• only one process at a time is **running** on the CPU • process gives up 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

[I] Scheduling: Definitions [14]

long-term scheduler	• **job scheduler** • which process on disk should be given memory? • result: new process in **ready queue** • important in batch systems • many processes in memory ⇒ high **degree of multiprogramming**
short-term scheduler	• **CPU scheduler** • which process in ready queue should be given CPU • result: new process on **CPU**

[I] Scheduling: Definitions [15]

CPU-bound	• most of its time doing computation - little I/O
I/O-bound	• most of its time doing I/O - little computation
multilevel scheduling	• classified into different groups • **foreground** (interactive) vs. • **background** (batch) • each group has its own ready queue

[I] Performance: Definitions [16]

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
through-put	• 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** • goal: fast turnaround

[I] Performance: Definitions [17]

response
- time waiting in the **ready queue** and
- executing on CPU until some output produced
- average is across all output events
- goal: fast response time

waiting
- sum of periods spent waiting in **ready queue**
- average is across all visits to ready queue
- goal: short waiting time

important
- **scheduler has a direct effect on waiting time**
- decides which process in queue gets to run next
- remaining processes must then wait longer
- OS cannot control code, amount of I/O, etc.

[I] Performance: Summary [18]

- 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})

[I] CPU Burst [19]

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

```
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  */
```

[I] Scheduling: FCFS [20]

- 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 [21]

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

PID	Burst
P_1	24
P_2	3
P_3	3

Gantt chart

P_1	P_2	P_3
0	24	27 → 30

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

PID	Burst
P_2	3
P_3	3
P_1	24

Gantt chart

P_2	P_3	P_1
0	3	6 → 30

- in general, **FCFS** average waiting time is not minimal
- in general, better to process shortest jobs first

[I] Scheduling: Round Robin (RR) [22]

- similar to **FCFS**, but **preemption** to switch between processes
- **time quantum** (**time slice**) is a small unit of time (10 to 100 ms)
- process is executed on the CPU for at most one time quantum
- implemented by using the **ready queue** as a **circular queue**
- **head** process gets the CPU
- uses less than a time quantum ⇒ gives up the CPU voluntarily
- uses full time quantum ⇒ **timer** will cause an **interrupt**
  - **context switch** will be executed
  - process will be put at the **tail** of queue

[I] Scheduling: RR [23]

- 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

PID	Burst
P_1	24
P_2	3
P_3	3

Gantt chart

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

- very large time quantum ⇒ **RR = FCFS**
- very small time quantum ⇒ context switch is too much overhead
- quantum ≈ CPU burst ⇒ 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 [25]

- 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

PID	Burst
P_1	6
P_2	8
P_3	7
P_4	3

Gantt chart

P_4	P_1	P_3	P_2	
0	3	9	16	24

- **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 [26]

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

$\alpha = 1 :\ \tau_{n+1} = t_n$

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

$\alpha = 1/2 :$ recent and past history the same

time	0	1	2	3	4	5	6	7
Burst (t_i)		6	4	6	4	13	13	13
Guess (τ_i)	10	8	6	6	5	9	11	12

$$\tau_1 = \frac{1}{2}t_0 + \frac{1}{2}\tau_0 = \frac{1}{2}6 + \frac{1}{2}10 = 8$$

[I] Scheduling: SJF [27]

- 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

PID	Burst	Arrival
P_1	8	0
P_2	4	1
P_3	9	2
P_4	5	3

Gantt chart

P_1	P_2	P_4	P_1	P_3	
0	1	5	10	17	26

- **nonpremptive SJF**: 7.75 ms

[I] Scheduling: Priority (PRIO) [28]

- assume a priority is associated with each process
- select highest priority process from the ready queue
- let τ 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 τ
- 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 [29]

- 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 ms

PID	Burst	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

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 [30]

- 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

[II] Critical Section: Example Problems [31]

```
program code: x=0; x++; print x;
     output: 1
```

- 3 processes execute same code with different output:

```
PID 1         PID 2         PID 3        x: (global variable)
x=0;                                     0
x++;                                     1
              x=0;                       0
                            x=0;         0
printf x;                                0 <=== output
              x++;                       1
              printf x;                  1 <=== output
                            x++;         2
                            printf x;    2 <=== output
```

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

```
PID 1                      PID 2
fetch i = 0 from memory
load on register
time-slice                 fetch, load, increment, store 1 in memory, slice
increment i
store 1 in memory
```

[II] Critical Section: Attempted Solution [32]

Shared Variable:

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

Code: At $p_i \in \{1, ..., n\}$:

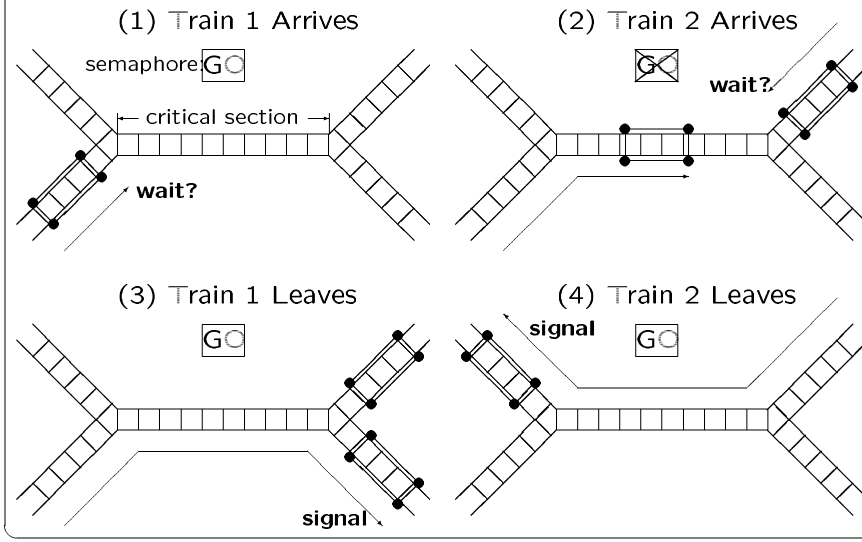
```
while (1) {

  /* 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 [33]

- 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 **semphore queue** (no **busy waiting**)
  - process is finished: **head** of FIFO queue enters section

[II] Semaphore: wait and signal [34]



[II] Semaphore: wait and signal [35]

- critical section: general solution

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

- critical section: example solution

```
while(1) {
   pm_wait(sem);
   x=0;
   x++;
   printf x;
   pm_signal(sem);
}
```

[II] Semaphore: wait and signal [36]

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

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

- **count** $\geq 0 \Rightarrow$ queue is **empty**
- **count** of **negative** $n \Rightarrow$ 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) }
```

[II] Semaphore: Initialization [37]

- 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**:

```
sem = pm_seminit(1);
```

- by setting **count=0** and then **signal**:

```
sem = pm_seminit(0);
pm_signal(sem);
```

[II] Semaphore: Count [38]

```
             PID 1                     PID 2                count
             sem=pm_seminit(1);                               1
             pm_wait(sem);                                    0
                 |                     pm_wait(sem);         -1
    (critical)   |
                 v
             pm_signal(sem);                                  0
             pm_wait(sem);                 |                 -1
                                           | (critical)
                                           v
                                       pm_signal(sem);        0
                 |                     pm_wait(sem);         -1
    (critical)   |
                 v
             pm_signal(sem);                                  0
```

[II] Critical Section: Deadlock [39]

- **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 [40]

- 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);
```

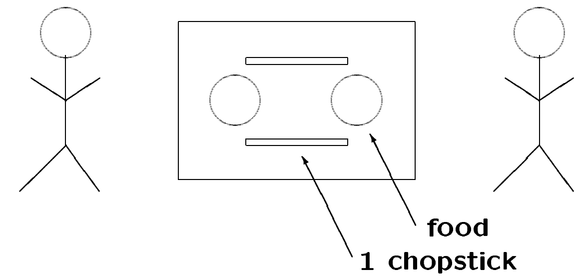
- both processes are **deadlocked** as each are **waiting** for the other

[III] Classical Problems: Introduction [41]

- **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 [42]

problem



- 2 philosophers
- 2 chopsticks (1 pair)

questions

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

answer

- TAKE TURNS USING RESOURCES

[III] Dining Philosophers [43]

Rule 1: a **philosopher** thinks.
Rule 2: a philosopher gets 2 chopsticks (1 pair).
Rule 3: a philosopher eats.
Rule 4: there are only 2 chopsticks (1 pair).
Rule 5: after eating, philosopher puts down 2 chopsticks (1 pair).

OK

Philosopher 1 uses the 2 chopsticks (1 pair) to eat. Then Philosopher 2 uses the 2 chopsticks (1 pair) to eat. Both make **progress** and the access to the **resources** (chopsticks) is **synchronized**.

BAD

Philosopher 1 gets one (1) chopstick. Philosopher 2 gets the other (1) chopstick. Neither can eat. Both are **deadlocked**.

[III] Dining Philosophers [44]

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

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

[III] Burns' Dining Philosophers [45]

Shared Variables:

1. *FORK*: semaphore array $[0..n-1]$, initially all available

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

```
do forever
  if even(i) then {
    wait(FORK_{i+1})   /* left fork */
    wait(FORK_i)   /* right fork */
  }
  if odd(i) then {
    wait(FORK_i)   /* right fork */
    wait(FORK_{i+1})   /* left fork */
  }
  /* Critical region */
  signal(FORK_i)
  signal(FORK_{i+1})
```

[III] Producer/Consumer [46]

problem

Producer:		Consumer:
	buffer	
data ⇒	[| | | |]	⇒ data

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

[III] Producer/Consumer [47]

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 [48]

- 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**

```
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 [49]

problem

- many readers OK: FILE ⇒ data
- one writer OK: FILE ⇐ data
- read/write NOT OK: data ⇒ FILE ⇒ 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 [50]

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.

[III] Readers/Writer [51]

- reader **waits** for writer to finish writing
- reader **signals** to writer when all readers are finished reading

```
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

```
while(1) {
   pm_wait(sem_wrt);
   /* write to file */
   pm_signal(sem_wrt);
}
```

[III] Gantt: Readers/Writer [52]

```
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

    | R1:W(SR) | R1:RC++ | R1:W(SW) | R2:W(SR)* | W1:W(SW)* | W2:W(SW)* |
RC:0           1
SR:1  0                             -1
SW:1                    0                      -1          -2
                                    R2->SRQ    W1->SWQ     W2->SWQ

    | R1:S(SR) | R1:W(SR)* | R2:RC++ | R2:S(SR) | R2:W(SR)* | R1:RC-- |
RC:                         2                               1
SR:   0          -1                    0          -1
SW:
      R2->RDQ    R1->SRQ               R1->RDQ    R2->SRQ

    | R1:S(SR) | R2:RC-- | R2:S(SW) | R2:S(SR) | W1:S(SW) | W2:S(SW)
RC:              0
SR:   0                             1
SW:                       -1                   0          1
      R2->RDQ             W1->RDQ              W2->RDQ
```

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

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

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 [57]

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 [58]

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 [59]

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 [60]

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 bearning 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.

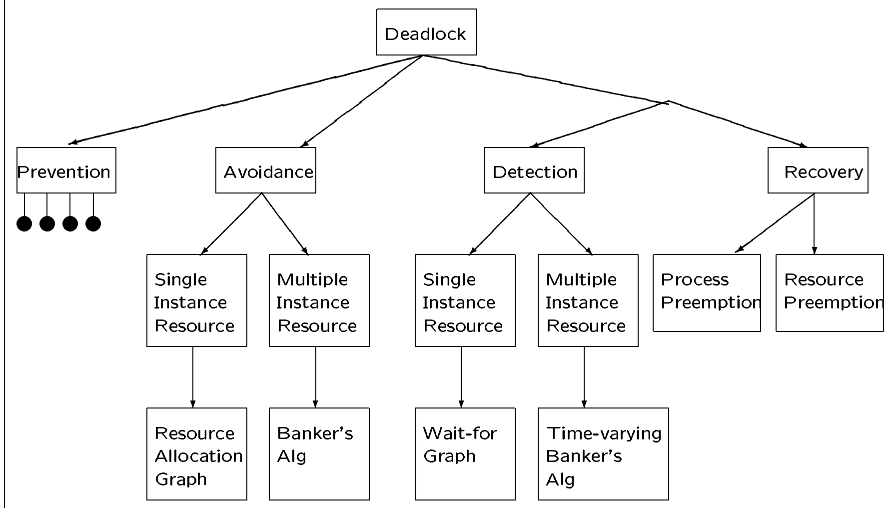
[V] Deadlock [61]

- 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 [62]



[V] Methods for Handling Deadlock [63]

- 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)

[V] Prevention: 4 Necessary Conditions [64]

- 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 [65]

- 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 [66]

- 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 [67]

- 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?

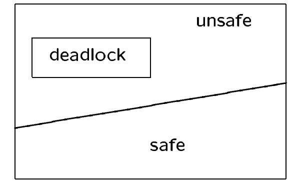
[V] Recovery [68]

- **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

[V] Avoidance [69]

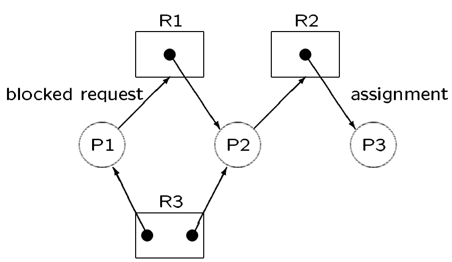
- processes give **advance** notice about **maximum** usage of resources
- processes make actual **requests** when they need a resource
- avoidance algorithm: **allocate** request only if it yields a **safe state**
  a safe sequence (SS) of processes exists
  such that each process can still get their 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

[V] Safe vs. Unsafe [70]



- 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

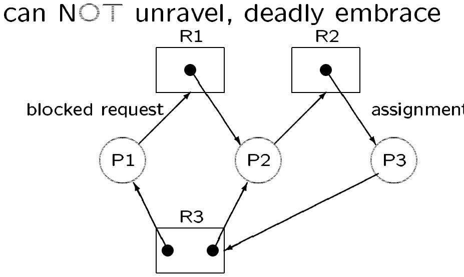
[V] Resource-Allocation Graph [71]



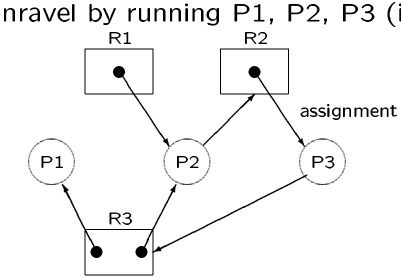
- process P1 blocked on resource R1
- R1 assigned to P2
- note that R3 has multiple instances (two)
- no cycle ⇒ no deadlock
- deadlock ⇒ cycle
- cycle ⇒ **maybe** deadlock
- single instance ∧ cycle ⇒ deadlock
- multiple instances ∧ cycle ⇒ do not know, use Banker's instead

[V] Avoidance with Multiple Instances [72]

DEADLOCK: can NOT unravel, deadly embrace



NO DEADLOCK: unravel by running P1, P2, P3 (i.e. SS)

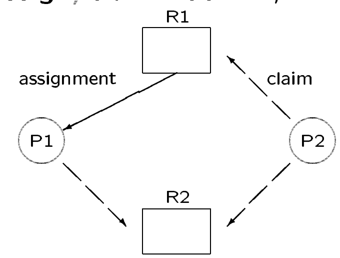


[V] Single vs. Multiple Instances [73]

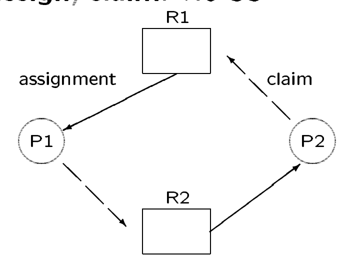
- both examples have cycles of **assign/blocked request**
- multiple instances: cycles don't tell us anything
- that's why we need the Banker's Algorithm
- but with Single Instances:
- cycle of **assign/blocked request** implies deadlock
- but avoidance does not want to get to this state
- need to stay 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

[V] Avoidance with Single Instances [74]

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

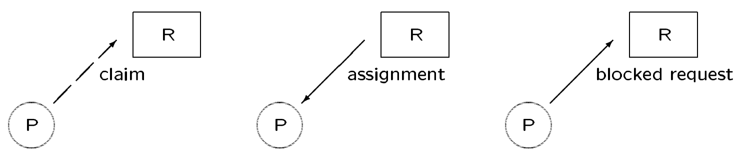


UNSAFE: cycle of **assign/claim**. No SS



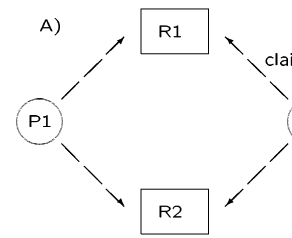
[V] Detailed Example [75]

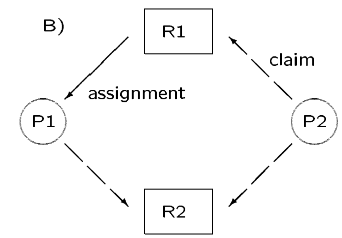
Consider these graph edges (do not follow Silberschatz):



- **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)

[V] Initial Claims/Assignments [76]

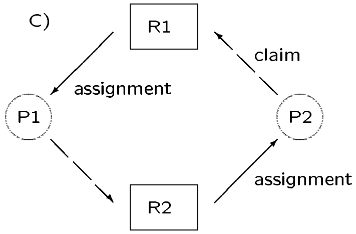


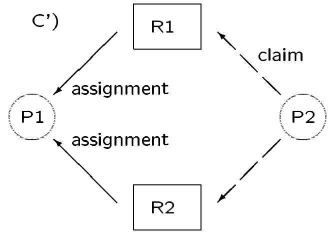


- Initial State A
- P1, P2 both claim R1, R2
- **might** need these resourses
- 2 SS: P1, P2 and P2, P1
- no cycles
- if necessary, they could finish
- same as Dining Philosophers

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

[V] Unsafe vs. Safe [77]

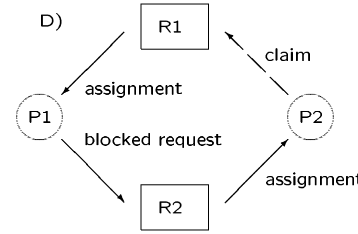


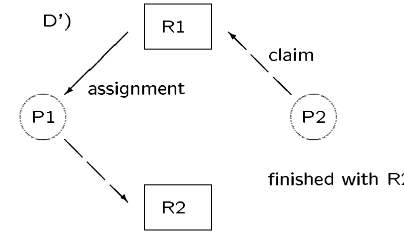


- State C could follow State B
- P2 requests R2, just assign it
- Bad Philosophers: each gets one R
- **cycle** (assign/claim edges)
- but NOT deadlock
- no SS, hence no guarantee of future
- Avoidance: UNDO, do NOT assign
- always stay safe, no deadlock

OR: C' follows B
P1 requests R2, assign
Good Philopsophers
no cycle
SS: P1, P2 obviously
safe, go ahead

[V] Trouble Brewing [78]

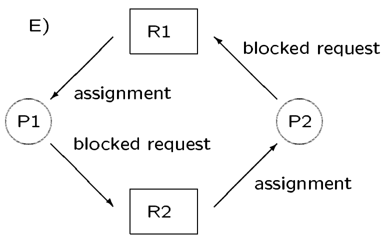


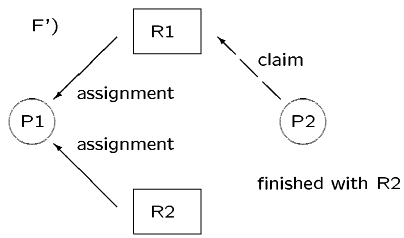


- if we do NOT UNDO C:
- i.e. not using Avoidance
- State D could follow C
- P1 requests R2, block
- guess what's coming?

OR: D' could follow C
P2 finished with R2

[V] Do Not Know Future [79]





- 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

[V] Avoidance: Conclusions [80]

- 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 resourse utilization, but guaranteed no deadlock

[V] Banker's Algorithm [81]

- 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 the allocation leaves system in safe state
  grant the resources
- otherwise
  suspend process until other processes release enough resources

[V] Example of Avoidance [82]

- 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)

	Max Need	–	Current Alloc	=	Need	Available
P_0	10		5		5	3
P_1	4		2		2	
P_2	9		2		7	

[V] Safe Sequence [83]

- 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:

	Max Need	–	Current Alloc	=	Need	Available
P_1	4		4		0	1

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

	Max Need	–	Current Alloc	=	Need	Available
P_1	4		0		0	5

- 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 Need $\leq$Available?
- if yes, then dump their Alloc into Available, repeat.
- if no, then no safe sequence (SS)

[V] Proof of Safe Sequence [84]

	Max Need	–	Current Alloc	=	Need		Available
P_1	4		2		2	$\leq$	3
P_0	10		5		5		
P_2	9		2		7		
P_1	4		0		0		
P_0	10		5		5	$\leq$	5
P_2	9		2		7		
P_1	4		0		0		
P_0	10		0		0		
P_2	9		2		7	$\leq$	10
P_1	4		0		0		
P_0	10		0		0		
P_2	9		0		0		12

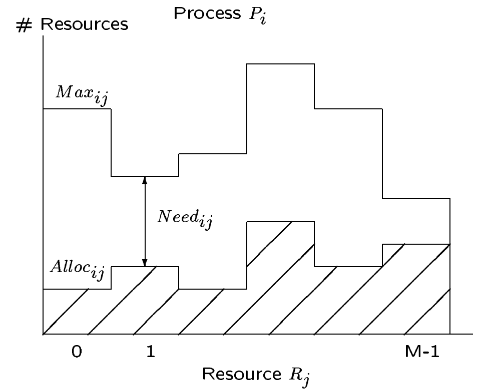
[V] No Safe Sequence [85]

- 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:

	Max Need	–	Current Alloc	=	Need	Available
P_0	10		5		5	2
P_1	4		2		2	
P_2	9		3		7	

- 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!**

[V] Many Resources/Multiple Instances [86]



- 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

[V] Banker: Data Structures [87]

```
#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 $X[i] \leq Y[i]$ 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 $Need_i$

[V] Safety Algorithm [88]

- 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**

[V] Banker: Safety Algorithm [89]

```
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
```

[V] Full Banker's Algorithm [90]

```
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
```

[V] State Example [91]

Initially:

```
Available
  A B C
 10 5 7
```

Later Snapshot:

```
      Max   -  Allocation  =  Need   Available
     A B C       A B C       A B C     A B C
P0   7 5 3       0 1 0       7 4 3     3 3 2
P1   3 2 2       2 0 0       1 2 2
P2   9 0 2       3 0 2       6 0 0
P3   2 2 2       2 1 1       0 1 1
P4   4 3 3       0 0 2       4 3 1
```

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

[V] State Example [92]

- 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 <= Available
- does not matter which one appears first in safe sequence
- both just increase the "working" available

[V] Safety Example [93]

```
     Max    -  Allocation  =  Need <= Work   Available
    A B C       A B C       A B C             A B C
P1  3 2 2       2 0 0       1 2 2 <= 3 3 2 <- 3 3 2
                  ----------------> +2 0 0
                                     -----
P3  2 2 2       2 1 1       0 1 1 <= 5 3 2
                  ----------------> +2 1 1
                                     -----
P4  4 3 3       0 0 2       4 3 1 <= 7 4 3
                  ----------------> +0 0 2
                                     -----
P2  9 0 2       3 0 2       6 0 0    7 4 5
                  ----------------> +3 0 2
                                     -----
P0  7 5 3       0 1 0       7 4 3 <=10 4 7
                  ----------------> +0 1 0
                                    ------
                                    10 5 7<<< initial system
```

[V] Full Banker Example [94]

- Say P_1 requests (1,0,2)
- Compare to $Need_1$: (1,0,2) $\leq$ (1,2,2)
- Compare to $Available$: (1,0,2) $\leq$ (3 3 2)
- Pretend to allocate resources:

```
      Max    -  Allocation  =  Need    Available
     A B C       A B C        A B C     A B C
P0   7 5 3       0 1 0        7 4 3     2 3 0<<<
P1   3 2 2       3 0 2<<<     0 2 0<<<
P2   9 0 2       3 0 2        6 0 0
P3   2 2 2       2 1 1        0 1 1
P4   4 3 3       0 0 2        4 3 1
```

- Is this safe? Yes: $< P_1, P_3, P_4, P_0, P_2 >$
- proof follows

[V] Banker Safety [95]

```
     Max    -  Allocation  =  Need <= Work   Available
    A B C       A B C       A B C    A B C    A B C
P1  3 2 2       3 0 2       0 2 0 <= 2 3 0 <- 2 3 0
                  ----------------> +3 0 2
                                     -----
P3  2 2 2       2 1 1       0 1 1 <= 5 3 2
                  ----------------> +2 1 1
                                     -----
P4  4 3 3       0 0 2       4 3 1 <= 7 4 3
                  ----------------> +0 0 2
                                     -----
P0  7 5 3       0 1 0       7 4 3 <= 7 4 5
                  ----------------> +0 1 0
                                     -----
P2  9 0 2       3 0 2       6 0 0 <= 7 5 5
                  ----------------> +3 0 2
                                     -----
                                    10 5 7 <<< initial system
```

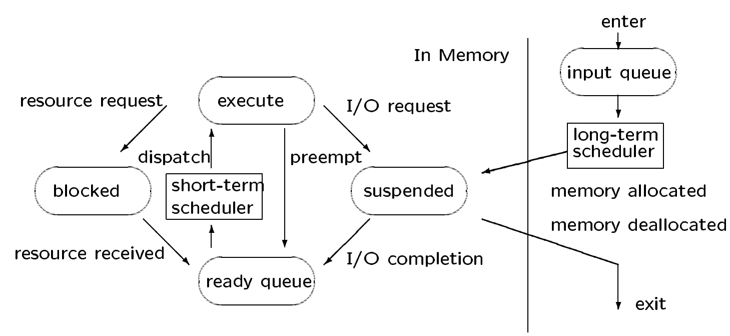
[V] More Examples [96]

- 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 [97]

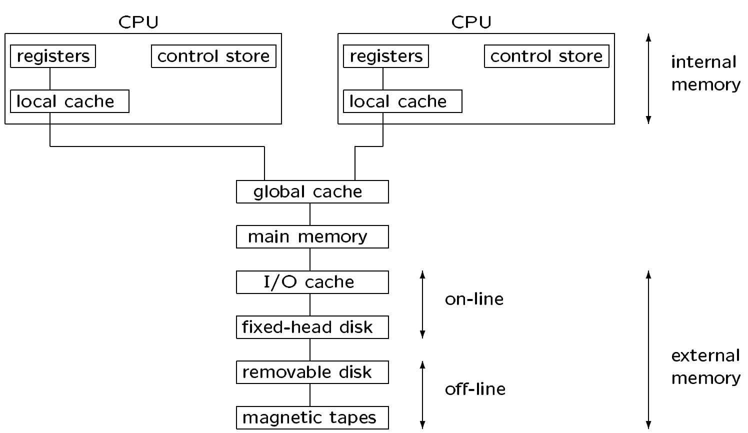
- CPU runs program instructions only when program is in memory
- programs do I/O sometimes ⇒ CPU wasted
- solution: **multiprogramming** (multitasking)
  - multiple programs (processes) share the memory
  - one program, at at time, gets CPU
  - simultaneous resource possession (CPU and memory)
  - better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

[VI.A] Long/Short Term Schedulers [98]



- 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*

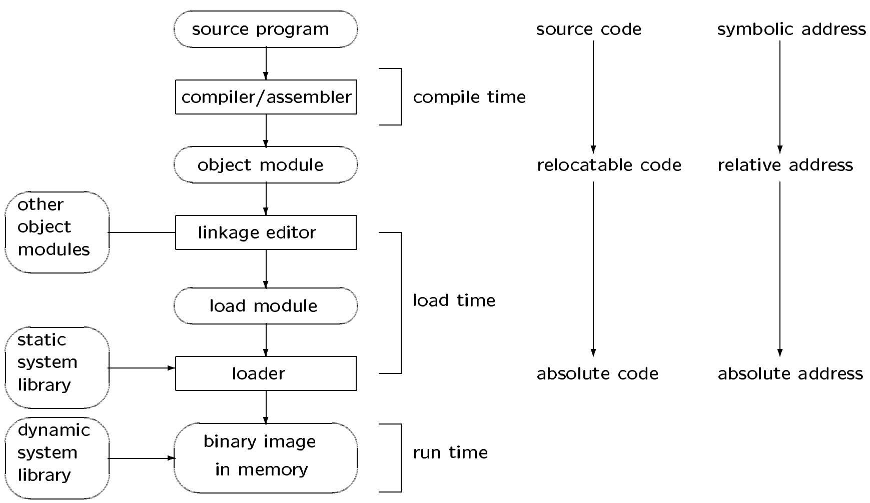
[VI.A] Memory: Storage Hierarchy [99]



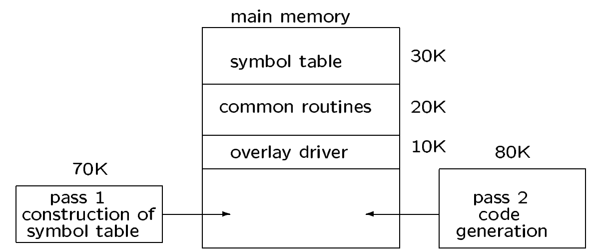
[VI.A] Memory: Devices [100]

memory type	capacity	access time	technology
control store	1K-32K words	3-100 ns	SRAM, ROM
registers	8-256 words	3-10 ns	SRAM
CPU cache	8 KB - 1 MB	3-100 ns	SRAM,DRAM
main memory	128 KB - 4 GB	20-200 ns	DRAM
I/O cache	32 KB - 1 MB	20-200 ns	DRAM
disk drive	1 MB -100 GB	10-65 ms	magnetic disk
tape drive	20 MG or more	seconds	magnetic tape

[VI.A] Memory: Address Binding [101]

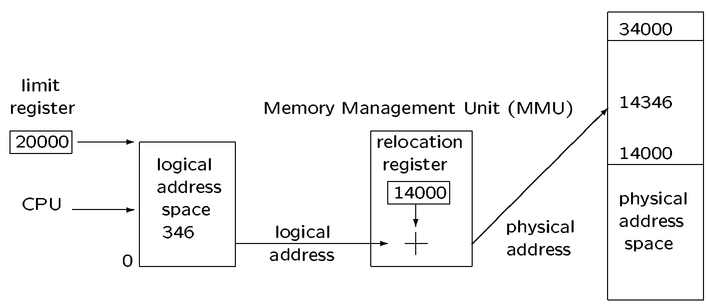


[VI.A] Memory: Overlays [102]



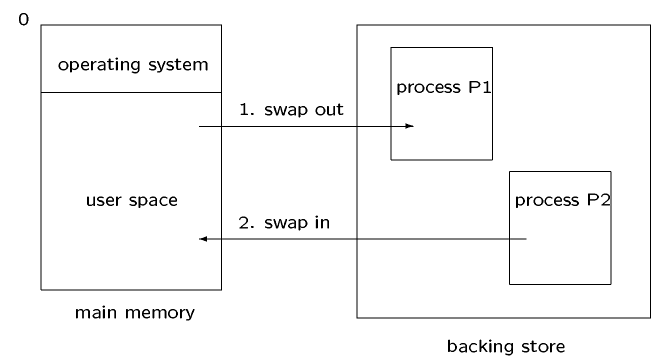
- program does pass 1; then pass 2
- programmer controls memory
- used when physical memory size was limited

[VI.A] Memory: Dynamic Relocation [103]



- 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 [104]



- swap-out: executing job (round-robin)
- swap-in : old job (maybe dynamic relocation)
- roll-out,roll-in: low priority for high priority
- 2 jobs × (100K size × 1000K/sec + 8ms latency) = 216ms
- quantum: much larger than 216ms

[VI.A] Memory: Contiguous Allocation [105]

- 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 [106]

	job queue	
process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15
TOTAL		58

Scheduling Discipline:

Job: FCFS

CPU: Round-Robin (Quantum=1)

Memory: 2560 KB

Dynamic Storage Allocation Strategy: Best fit

[VI.A] Memory: Scheduling Details [107]

time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P_1	1	1	1	2	2	2	3	3	3	4	4	4	5	5
P_2	0	1	1	1	2	2	2	3	3	3	4	4	4	5
P_3	0	0	1	1	1	2	2	2	3	3	3	4	4	4

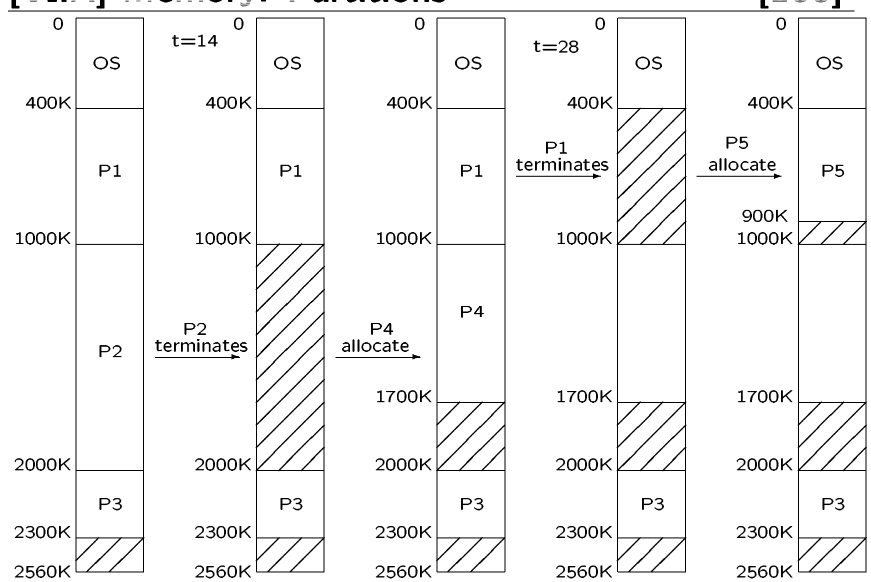
time	15	16	17	18	19	20	21	22	23	24	25	26	27	28
P_1	5	6	6	6	7	7	7	8	8	8	9	9	9	10
P_4	0	0	1	1	1	2	2	2	3	3	3	4	4	4
P_3	5	5	5	6	6	6	7	7	7	8	8	8	9	9

time	29	30	31	32	33	34	35	36	37	38
P_5	0	0	1	1	1	2	2	2	3	3
P_4	5	5	5	6	6	6	7	7	7	8
P_3	9	10	10	10	11	11	11	12	12	12

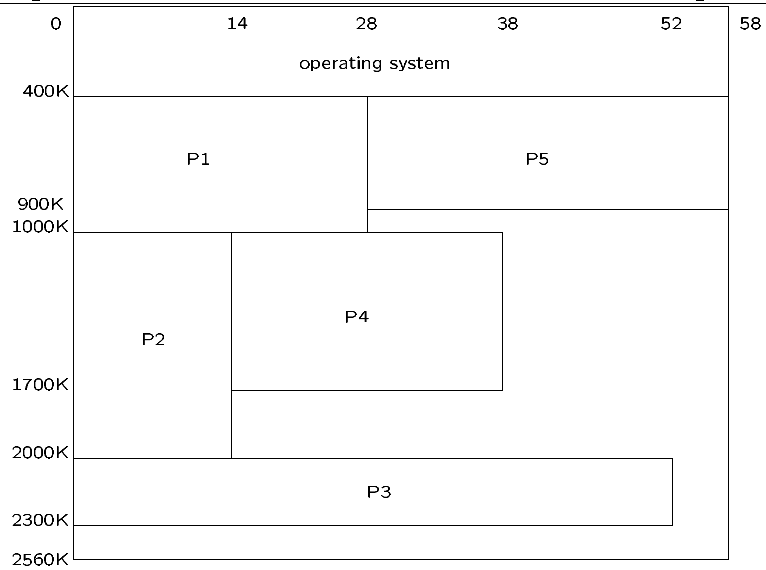
time	39	40	41	42	43	44	45	46	47	48	49	50	51	52
P_5	3	4	4	5	5	6	6	7	7	8	8	9	9	10
P_3	13	13	14	14	15	15	16	16	17	17	18	18	19	19

time	53	54	55	56	57	58
P_5	10	11	12	13	14	15
P_3	20					

[VI.A] Memory: Partitions [108]



[VI.A] Memory Usage vs. CPU Usage [109]



[V.A] Memory: Search Strategies [110]

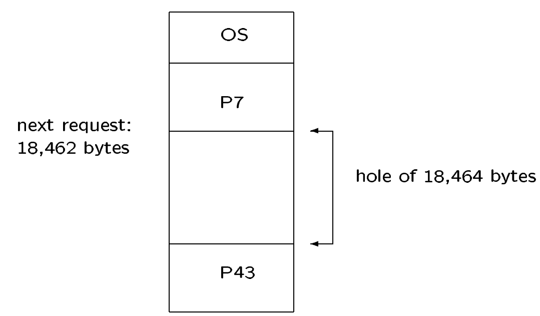
- list of memory holes: which one to use?
- first-fit : allocate the first hole which is big enough
- best-fit : allocate the smallest hole which is big enough
    - produces the smallest left-over hole
- worst-fit: allocate the largest hole
    - produces the largest left-over hole

strategy	search time	memory utilization
first-fit	fast	good
best-fit	slow	good
worst-fit	slow	bad

[VI.A] Memory: External Fragmentation [111]

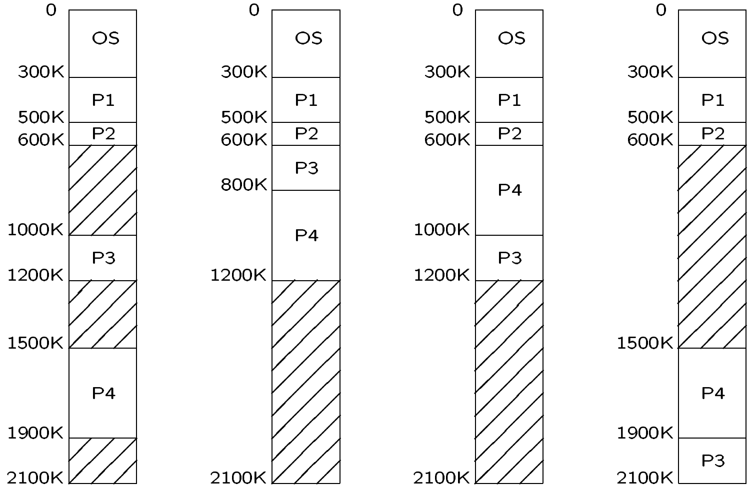
- request for memory cannot be satisfied
    - even though total free memory is sufficient
    - but not contiguous (broken into small holes)
- first-fit or best-fit may be better
- 50-percent rule:
    - given N allocated blocks $\Rightarrow$ 0.5N blocks unusable (1/3 wasted)
- solution: compaction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

[VI.A] Memory: Internal Fragmentation [112]



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

[VI.A] Memory: Compaction Techniques [113]

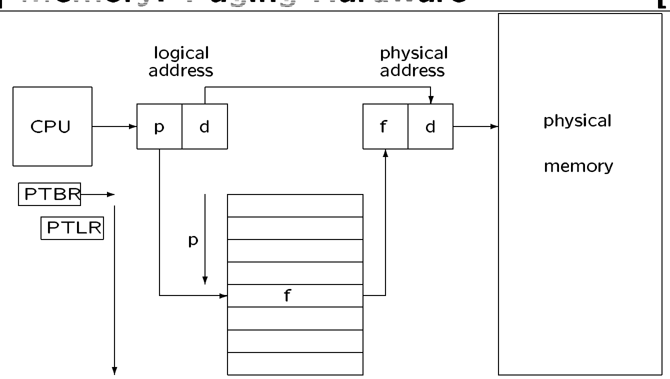


- move 600K, or 400K, or 200K
- dynamic relocation and swapping

[VI.A] Memory: Paging [114]

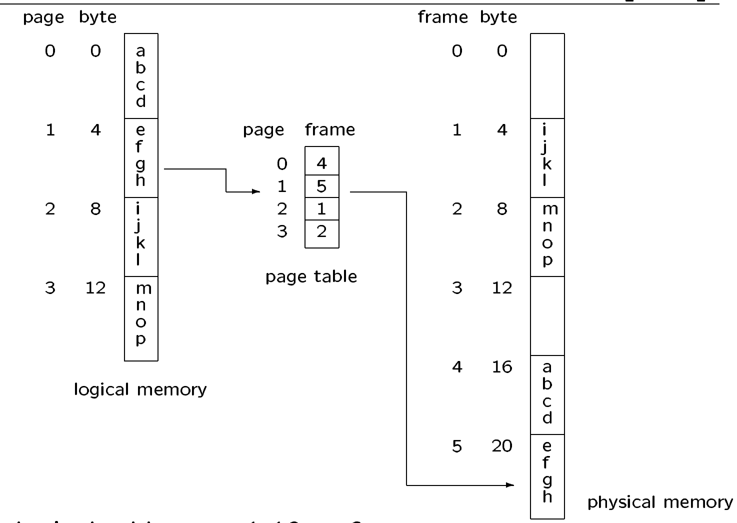
- goal: eliminate external fragmentation
- (allow noncontiguous processes)
- each process is a set of fixed-size pages
- pages are stored in same-size physical memory "frames"
- page table "connects" logical pages with physical frames
- may still have internal fragmentation

[VI.A] Memory: Paging Hardware [115]



- 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

[VI.A] Memory: Paging [116]



- byte 'g': logical address = 1 10 = 6
- byte 'g': physical address = 101 10 = 22

[VI.A] Memory: Logical Address [117]

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

page number	page offset
p	d
$m-n$	n

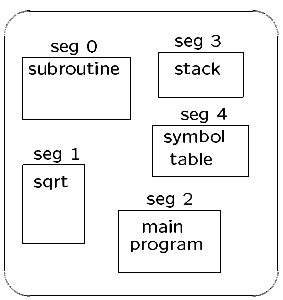
[VI.A] Memory: Logical Addresses [118]

- $n = 10$ and $m = 32$

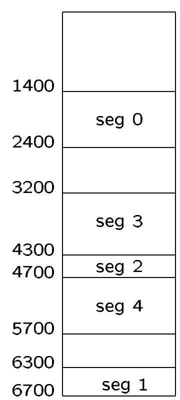
```
typedef union {
  struct {
    unsigned int page  :22;
    unsigned int offset:10;
  } logical;
  int addr;
} laddr;
laddr l;
```

- page 0: 0..1023
- page 1: 1024..2047
- page $2^{m-n} = 4,194,304$
- `l = 1026; /* page=1, offset=2 */`

[VI.A] Memory: Segmentation [119]



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700



- 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)

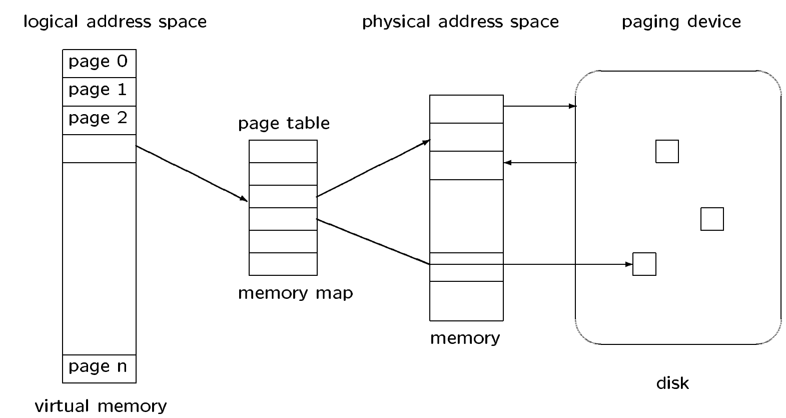
[V.A] Memory: Paging vs. Segmentation [120]

- 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 [121]

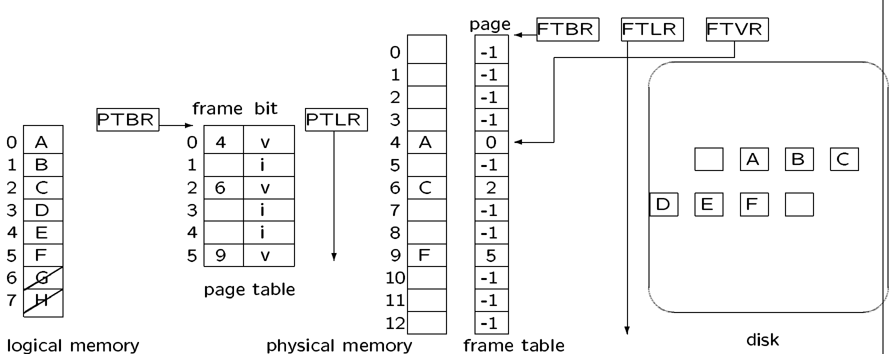
- up to now: all of a process in main memory (somewhere)
  - partitions, pages, segments
- now: virtual memory
  - allow execution of processes which may be partially in memory
- benefits:
  - programs can be large and memory can be small
  - increased multiprogramming ⇒ better performance
  - less I/O 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 [122]



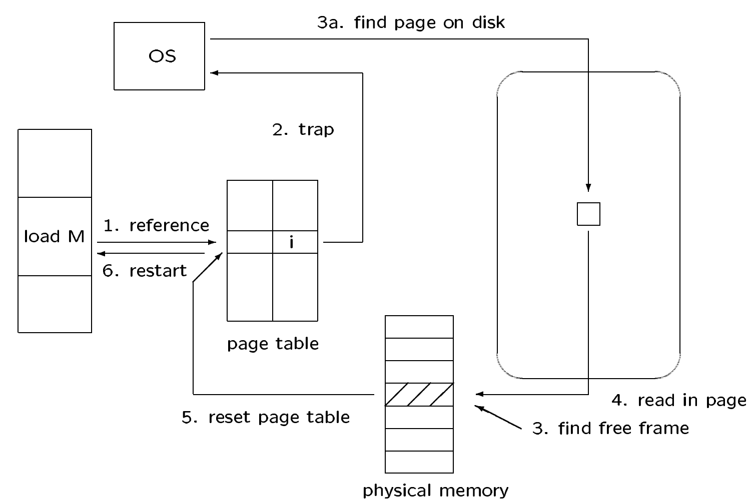
- logical address vs. physical address
- demand paging: only "necessary" pages are brought into memory

[VI.B] VM: Valid-Invalid Bit and Frame Table [123]



- 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 [124]



[VI.B] VM: Page Fault Details [125]

- 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 [127]

- "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 [128]

```
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 [130]

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

page:	faults:	frames:			time:		
7	1	7	-1	-1	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	5	3
3	5	2	0	3	4	5	6
0	5	2	0	3	4	7	6
4	6	4	0	3	8	7	6
2	7	4	0	2	8	7	9
3	8	4	3	2	8	10	9
0	9	0	3	2	11	10	9
3	9	0	3	2	11	12	9
2	9	0	3	2	11	12	13
1	10	1	3	2	14	12	13
2	10	1	3	2	14	12	15
0	11	1	0	2	14	16	15
1	11	1	0	2	17	16	15
7	12	1	0	7	17	16	18
0	12	1	0	7	17	19	18
1	12	1	0	7	20	19	18

[VI.B] VM: LRU_STACK [131]

- 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 [132]

page:	faults:	frames:			stack:		
7	1	7	-1	-1	7		
0	2	7	0	-1	7	0	
1	3	7	0	1	7	0	1
2	4	2	0	1	0	1	2
0	4	2	0	1	1	2	0
3	5	2	0	3	2	0	3
0	5	2	0	3	2	3	0
4	6	4	0	3	3	0	4
2	7	4	0	2	0	4	2
3	8	4	3	2	4	2	3
0	9	0	3	2	2	3	0
3	9	0	3	2	2	0	3
2	9	0	3	2	0	3	2
1	10	1	3	2	3	2	1
2	10	1	3	2	3	1	2
0	11	1	0	2	1	2	0
1	11	1	0	2	2	0	1
7	12	1	0	7	0	1	7
0	12	1	0	7	1	7	0
1	12	1	0	7	7	0	1

[VI.B] VM: LRU_REF [133]

- 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

↑

↑

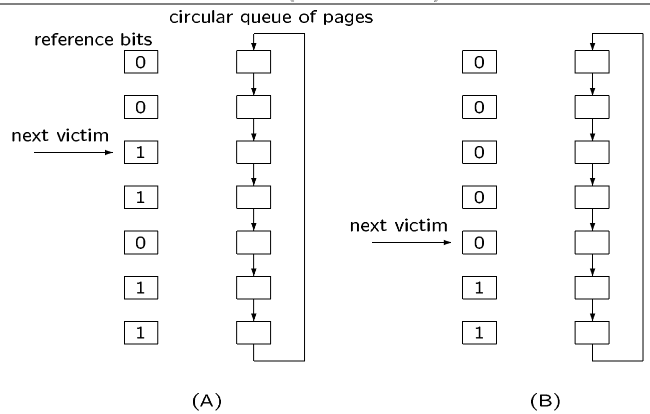
↑

- apply OPT algorithm: 2 0 3 → 4 0 3

[VI.B] VM: LRU Approximations [134]

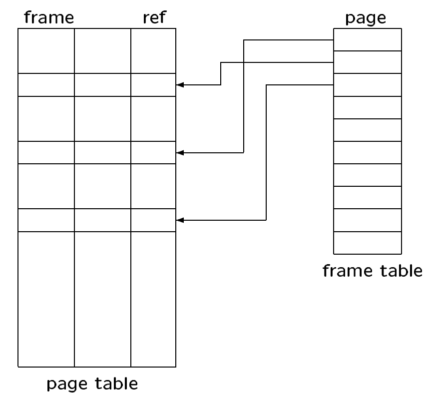
- hardware usually does not support LRU
- but does support REF bit
  - interrupt every 100 msec
  - move REF bit to 8-bit shift register (and clear)
  - 00000000 ⇒ no refs in last 8 periods
  - 11111111 ⇒ at least one ref in each period
  - 01111111 ⇒ no ref in the recent period
  - smallest integer ⇒ ≈ 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

[VI.B] VM: CLOCK Circular Queue [136]



- frame table is circular queue of pages

[V.B] VM: CLOCK [137]

- better than FIFO (15) but worse than LRU (12)

```
page: 7  faults: 1  frames:  7 -1 -1   ref:  1
      0          2           7  0 -1         1  1
      1          3           7  0  1         1  1  1
      2          4           2  0  1         1  0  0
      0          4           2  0  1         1  1  0
      3          5           2  0  3         1  0  1
      0          5           2  0  3         1  1  1
      4          6           4  0  3         1  0  0
      2          7           4  2  3         1  1  0
      3          7           4  2  3         1  1  1
      0          8           4  2  0         0  0  1
      3          9           3  2  0         1  0  1
      2          9           3  2  0         1  1  1
      1         10           3  1  0         0  1  0
      2         11           3  1  2         0  1  1
      0         12           0  1  2         1  1  1
      1         12           0  1  2         1  1  1
      7         13           0  7  2         0  1  0
      0         13           0  7  2         1  1  0
      1         14           0  7  1         1  1  1
```

[V.B] VM: ENHANCED Second Chance [138]

- 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
```

[VI.B] VM: ENHANCED [139]

- 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
wpage: 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 1,0 0,1
       3          5           2  0  3              1,0 0,0 1,0
wpage: 0          5           2  0  3              1,0 1,1 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 [140]

- 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

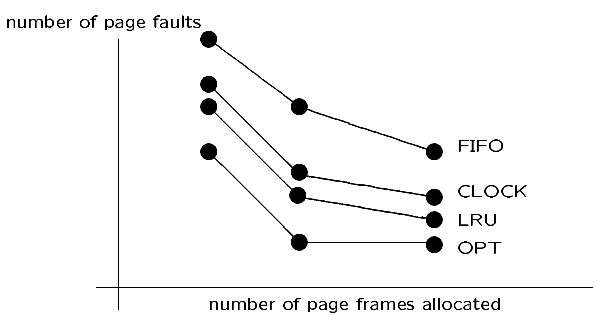
[VI.B] VM: LFU [141]

page:	faults:	frames:			count:		
7	1	7	-1	-1	1		
0	2	7	0	-1	1	1	
1	3	7	0	1	1	1	1
2	4	2	0	1	1	1	1
0	4	2	0	1	1	2	1
3	5	2	0	3	1	2	1
0	5	2	0	3	1	3	1
4	6	4	0	3	1	3	1
2	7	4	0	2	1	3	1
3	8	3	0	2	1	3	1
0	8	3	0	2	1	4	1
3	8	3	0	2	2	4	1
2	8	3	0	2	2	4	2
1	9	3	0	1	2	4	1
2	10	3	0	2	2	4	1
0	10	3	0	2	2	5	1
1	11	3	0	1	2	5	1
7	12	3	0	7	2	5	1
0	12	3	0	7	2	6	1
1	13	3	0	1	2	6	1

[VI.B] VM: MFU [142]

page:	faults:	frames:			count:		
7	1	7	-1	-1	1		
0	2	7	0	-1	1	1	
1	3	7	0	1	1	1	1
2	4	2	0	1	1	1	1
0	4	2	0	1	1	2	1
3	5	2	3	1	1	1	1
0	6	2	3	0	1	1	1
4	7	4	3	0	1	1	1
2	8	4	2	0	1	1	1
3	9	4	2	3	1	1	1
0	10	0	2	3	1	1	1
3	10	0	2	3	1	1	2
2	10	0	2	3	1	2	2
1	11	0	1	3	1	1	2
2	12	0	1	2	1	1	1
0	12	0	1	2	2	1	1
1	12	0	1	2	2	2	1
7	13	7	1	2	1	2	1
0	14	7	0	2	1	1	1
1	15	7	0	1	1	1	1

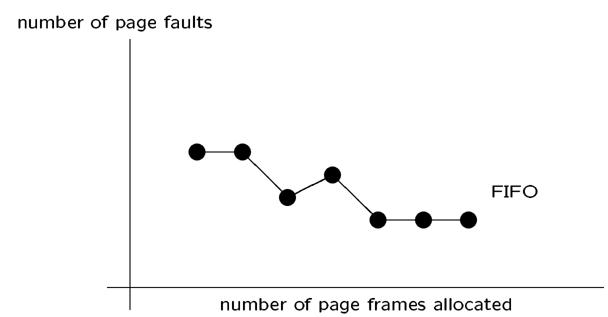
[VI.B] Replacement Algorithm Performance [143]



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

```
mem> mode 2
mem> init 8 8
```

[VI.B] VM: Belady's Anomaly [144]



- more frames ⇒ more faults
- FIFO may replace pages that are just about to be used again

[VI.B] VM: Belady's Anomaly [145]

```
    page: 1  faults: 1  frames:  1 -1 -1  time:  1
          2          2           1  2 -1         1  2
          3          3           1  2  3         1  2  3
          4          4           4  2  3         4  2  3
          1          5           4  1  3         4  5  3
          2          6           4  1  2         4  5  6
          5          7           5  1  2         7  5  6
          1          7           5  1  2         7  5  6
          2          7           5  1  2         7  5  6
          3          8           5  3  2         7 10  6
          4          9           5  3  4         7 10 11
          5          9           5  3  4         7 10 11
    page: 1  faults: 1  frames:  1 -1 -1 -1  time:  1
          2          2           1  2 -1 -1         1  2
          3          3           1  2  3 -1         1  2  3
          4          4           1  2  3  4         1  2  3  4
          1          4           1  2  3  4         1  2  3  4
          2          4           1  2  3  4         1  2  3  4
          5          5           5  2  3  4         7  2  3  4
          1          6           5  1  3  4         7  8  3  4
          2          7           5  1  2  4         7  8  9  4
          3          8           5  1  2  3         7  8  9 10
          4          9           4  1  2  3        11  8  9 10
          5         10           4  5  2  3        11 12  9 10
```

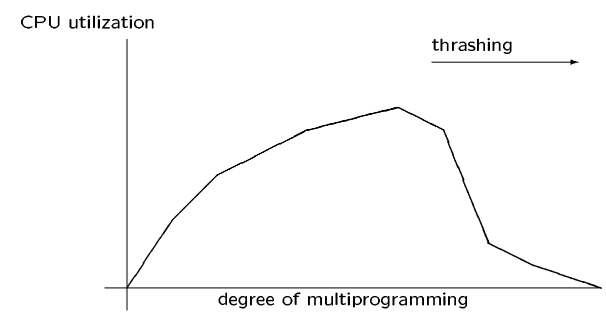
[VI.B] VM: Allocation of Frames [146]

- 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: $s_i/S \times m$
  - $S = \sum s_i$
  - 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 [147]

- 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 [148]

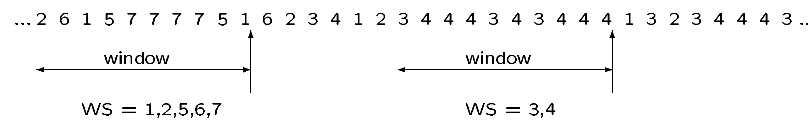


[VI.B] VM: Thrashing Solutions [149]

- 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 [150]

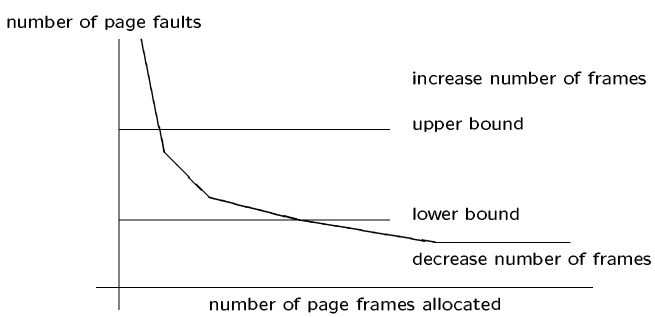
- approximation to the program's locality
- let $\triangle$ be the *working-set window*
- keep list of all pages used during the last $\triangle$ page references



- small $\triangle \Rightarrow$ does not encompass locality
- large $\triangle \Rightarrow$ 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

[VI.B] VM: Page-Fault Frequency Strategy [151]

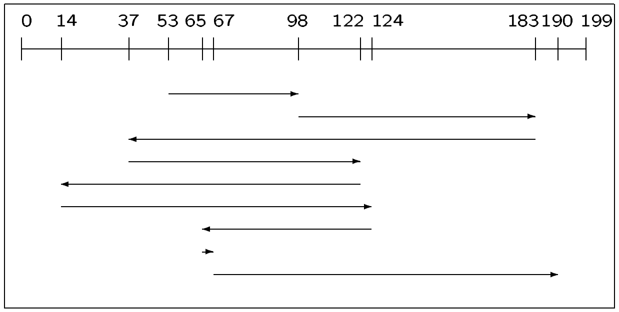
- PFF Strategy
- direct approach to solve thrashing
- may need to suspend other processes to get more frames



[VII] Disk Scheduling [152]

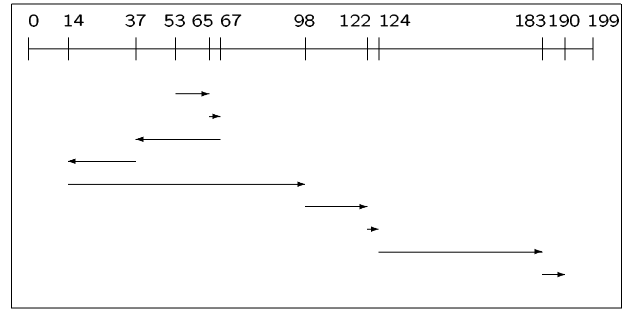
- physical device like disk is slow compared to CPU
- to access a block (say 512 bytes):
  - disk-arm moves to appropriate radius: **seek time** (11ms)
  - spinning disk (platter) rotates to position: **latency time** (13ms)
  - read-write head copies data: **transfer time** (53micro)
- during one read/write, many new block requests may arrive
- so OS knows in advance a batch of block requests
- how can a disk scheduling algorithm help improve the **seek time**?

[VII] Disk: FCFS [153]



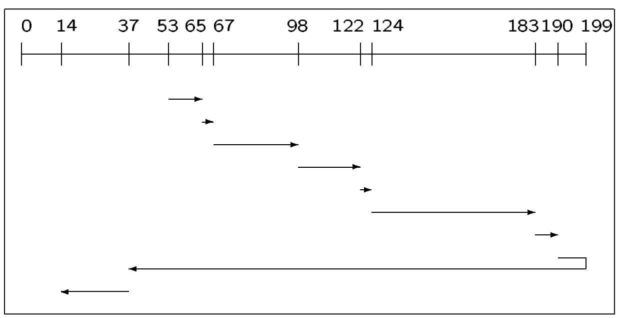
- First-Come First-Served: process blocks in the order of arrival
- while processing block 53, nine more requests arrive
- process the nine requests: 98, 183, 37, 122, 14, 124, 65, 67, 190
- long disk-arm movements back-and-forth, poor seek time
- total distance d = 763 (53..98..183..37..)
- not a good scheduling algorithm

[VII] Disk: SSTF [154]



- 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

[VII] Disk: SCAN [155]

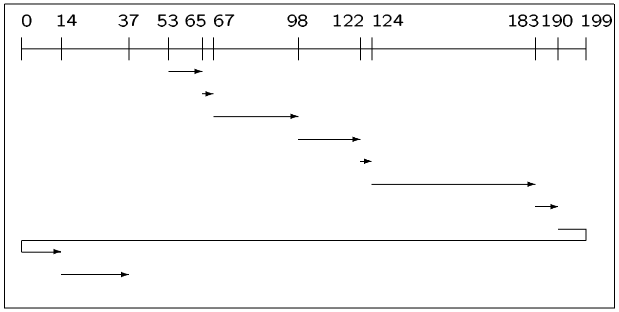


- SCAN is the first of 4 **elevator** algorithms
- like an elevator, move up/down (actually inside/outside)
- process people (block requests) as it goes 0..199..0..199..
- assume arm is moving upwards as it encounters 53
- keep on going all the way to end of disk 199, and back to 0
- 65, 67, 98, 122, 124, 183, 190, (199), 37, 14 (and then 0, repeat)
- d = 146 (53..199) + 185 (199..14) = 331

[VII] Disk: SCAN [156]

- 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

[VII] Disk: C-SCAN [157]

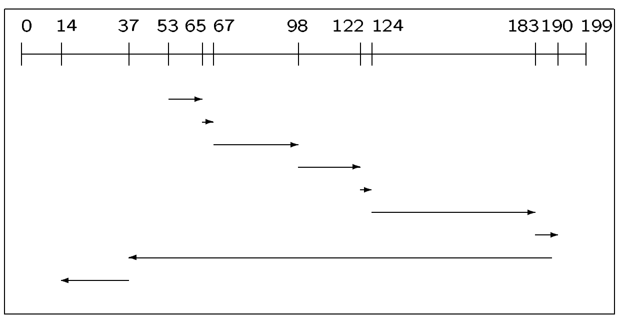


- C-SCAN: Circular-SCAN, do no process blocks on way down
- in this case, 14, 37 will be processed on the way back up
- 65, 67, 98, 122, 124, 183, 190, (199), (0), 14, 37
- d = 146 (53..199) + 199 (199..0) + 37 (0..37) = 382

[VII] Disk: C-SCAN [158]

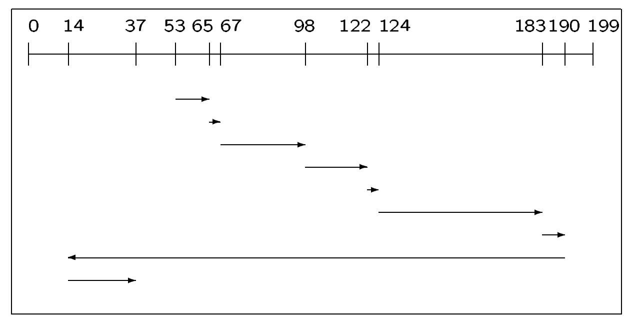
- what if 75 arrives while processing 124?
- it will be processed last, after 37, so fair
- what if 75 arrives while processing 67?
- it will be processeed right afterwards because arm is moving up
- it is processed before older 98, so still some unfairness
- SCAN algorithms move to edges even though no blocks to process
- these algorithms are meant as examples
- get rid of extra movement: LOOK

[VII] Disk: LOOK [159]



- 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

[VII] Disk: C-LOOK [160]



- 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 [161]